Für Barbara, Horst, Norbert und Sabine

Vorwort

Computerviren und andere Softwareanomalien wie trojanische Programme und elektronische Würmer haben seit 1986 mehr Aufsehen erregt als irgendein anderes Thema aus der Welt der Datenverarbeitung. Die Vorstellung, daß etwas mysteriöses, geradezu lebendiges Computer wie eine Krankheit befällt und Daten vernichtet, wurde von den Medien dankbar aufgenommen und ausgebaut. Darstellungen der Fähigkeiten von Computerviren und der Umfang der durch sie verursachten Schäden waren und sind oft maßlos übertrieben. Trotzdem stellen Softwareanomalien eine neue Art Gefahr dar, die niemand, der sich mit Datenverarbeitung befaßt, übersehen kann und darf.

Die Existenz einer Bedrohung der Daten- und Rechnersicherheit läßt eine Nachfrage nach Maßnahmen zur Abwehr von Softwareanomalien entstehen. In den Goldgräberjahren der Computerviren ab etwa 1988 kamen die ersten Abwehrprogramme auf den Markt. Mittlerweile ist die Situation fast unüberschaubar geworden. Die Anzahl der bekannten Viren hat die Tausendermarke überschritten, und es gibt Dutzende Programme zu ihrer Abwehr. Auch bei den Printmedien hat sich einiges getan. Das große Publikumsinteresse auf breiter Front hat die unterschiedlichsten Veröffentlichungen hervorgebracht. Nach ersten frühen Artikeln in Computerzeitschriften kam ab etwa 1989 eine ganze Anzahl Broschüren und Bücher heraus.

"Warum also noch ein Buch über Computerviren?", so könnte man fragen. Das aktuelle Spektrum reicht von Publikationen, die das Thema eher unterhaltsam behandeln bis hin zu Büchern für den Profi, der durch seinen Beruf für die Sicherheit von Rechneranlagen verantwortlich ist. Die meisten Titel befassen sich mit konkreten Viren und Antivirusprogrammen, wobei die Palette Betriebsanleitungen zu kommerzieller Software ebenso umfaßt wie die Virusprogrammierung inklusive Listing zum Abtippen.

Also was ist neu an *diesem* Buch? Es ermöglicht dem Leser, die Bedrohung durch Softwareanomalien zu erkennen und einzuschätzen, die Funktion der unterschiedlichen Anomalie-Typen zu verstehen und aus der Analyse charakteristischer Eigenschaften Methoden zur ihrer Abwehr zu entwickeln. Diese Überlegungen sind noch theoretischer Natur und damit betriebssystemunabhängig und universell anwendbar. Ein MS-DOS-spezifischer Teil vermittelt dem Leser Kenntnisse der Systemprogrammierung unter MS-DOS. Dabei wird der Aufbau und die Arbeitsweise des Betriebssystems in seiner Funktion als Basis für Computerviren ausführlich untersucht. Hier beginnt der eigentlich neuartige Teil des Buches, die Entwicklung von eigenen Abwehrprogrammen auf der Basis von zuvor erarbeiteten theoretischen Grundlagen. Obwohl es dabei bis auf unterste Systemebenen hinabgeht, versetzt das stufenweise Vorgehen jeden in die Lage, die Entwicklung der Programme mitzuverfolgen und zu verstehen. Voraussetzung sind Grundkenntnisse in "C".

Die in diesem Buch entwickelten Funktionsbibliotheken und Programme sind praxistauglich und können als Basis für eigene Weiterentwicklungen dienen. Auf der Programmdiskette sind die dokumentierten Quelltexte zu allen Funktionen und Programmen dieses Buches enthalten. Dazu kommen gebrauchsfertig übersetzte Bibliotheken und Programme sowie Texte, die sich mit der Systemprogrammierung unter MS-DOS, der Abwehr von Computerviren und dem Zugriff auf öffentliche Netze befassen.

Auch wenn die Ambitionen des Lesers nicht in Richtung Softwareentwicklung gehen, so werden doch grundlegende Mechanismen vorgestellt und implementiert, wie sie fast alle kommerziell verfügbare Antivirusprogramme verwenden. Diese Kenntnisse ermöglichen es, diese Produkte zur beurteilen und zu einer solide fundierten Entscheidung zu gelangen. Auf eine konkrete Marktübersicht wird verzichtet, weil das Angebot zum einen sehr reichhaltig ist und sich zum anderen, ebenso wie die Bedrohungssituation, ständig verändert.

Zum Zustandekommen dieses Buches haben mehrere Personen beigetragen. Angefangen hat alles mit dem Konzept der kontrollierten Isolation, das von Prof. Dr. Siegmar Groß von der FH Fulda stammt. Diese Methode des Schutzes von Computern vor der Ausführung unzulässiger Programme wurde das Thema meiner Diplomarbeit, die Herr Groß zusammen mit Prof. Dr. Ing. Werner Heinzel betreut hat. An dieser Stelle sei noch einmal Barbara Foltin und Sabine Müller gedankt, die unermüdlich Berge von Text korrigiert haben und so den Abgabetermin retteten. Horst Kratz, Norbert Röstel und viele andere trugen dazu bei, daß das Studium insgesamt eine ebenso lebenswerte wie erfolgreiche Sache wurde.

Herr Heinzel als Herausgeber der Reihe PC professionell machte es schließlich möglich, daß aus der stark erweiterten Diplomarbeit dieses Buch werden konnte. Gesetzt wurde der Text auf einem 386er AT unter MS-DOS mit dem Satzsystem emTEX und dem Makropaket GLATEX. Ralph Babel und Herr Hoffman (DTM) halfen bei der Drucklegung. Meine Eltern korrigierten den gesamten Text und sorgten dafür, daß ich ungestört als Autor arbeiten konnte.

Taunusstein, im Dezember 1991 Christian Treber

Inhaltsverzeichnis

1 Theorie der Softwareanomalien **3**
- 1.1 Würmer . . . 4
  - 1.1.1 Definition . . . 4
  - 1.1.2 Trapdoors, triviale Paßwörter, Bugs: Der INTERNET-Worm . . . 4
- 1.2 Trojaner . . . 6
  - 1.2.1 Definition . . . 6
  - 1.2.2 Mangelnde Vorsicht: Der AIDS-Trojaner . . . 6
- 1.3 Computerviren . . . 7
  - 1.3.1 Definition . . . 7
  - 1.3.2 Transport (Vektor) . . . 7
  - 1.3.3 Aktivierung und Residenz . . . 8
  - 1.3.4 Infektionsmechanismus (Typologie) . . . 11
  - 1.3.5 Tarnung . . . 14
- 1.4 Die Bedrohung durch Softwareanomalien . . . 15
  - 1.4.1 Denial of Service . . . 15
  - 1.4.2 Software-Schäden . . . 15
  - 1.4.3 Hardware-Schäden . . . 16
- 1.5 Historie und Dynamik der Entwicklung . . . 17
  - 1.5.1 Die Experimente und Theorien von Fred Cohen . . . 17
  - 1.5.2 Die Verbreitung von Viren bis heute . . . 18
- 1.6 Maßnahmen öffentlicher Stellen . . . 20
  - 1.6.1 Die Rechtslage in Deutschland . . . 21
  - 1.6.2 Die Rechtslage in den USA . . . 21
  - 1.6.3 Sicherheit von IT-Systemen . . . 22
  - 1.6.4 Organisationen zur Virusbekämpfung . . . 24

2 Theorie der Abwehr **25**
- 2.1 Analyse: Ursachen der Verseuchung . . . 25
  - 2.1.1 Motive der Programmierer . . . 26
  - 2.1.2 Verbreitungswege . . . 28
  - 2.1.3 Motive der Anwender . . . 30
- 2.2 Konventionelle Sicherheitsmaßnahmen . . . 31
  - 2.2.1 Organisation (Kontrollmaßnahmen) . . . 31
  - 2.2.2 Grundlegende Maßnahmen . . . 36

2.2.3 Ethik . . . 38
2.3 MS-DOS und andere Betriebssysteme . . . 38
2.3.1 Innere Sicherheit (Zugriffskontrolle) . . . 39
2.3.2 Äußere Sicherheit (Zugangskontrolle) . . . 41
2.4 Konzepte zur Virenabwehr (Cohen's Theorien) . . . 44
2.5 Kommerziell verfügbare Konzepte . . . 46
2.5.1 Überwachung (Watcher) . . . 46
2.5.2 Detektion (Scanner) . . . 48
2.5.3 Schutz der Integrität (Checker) . . . 50
2.6 Analogien zur Biologie . . . 55
2.7 Alternative Konzepte . . . 59
2.7.1 Schutzzonen und Kontrollpunkte . . . 59
2.7.2 Generelle Verbote . . . 60
2.7.3 Strikte Isolation . . . 61
2.7.4 Kontrollierte Isolation . . . 61
2.7.5 Offenes System (Explizite Validierung) . . . 62
2.7.6 Offenes System (Wahrung der Integrität) . . . 62
2.7.7 Zusätzliche Maßnahmen . . . 63
2.7.8 Vergleich der Konzepte . . . 64

3 Systemprogrammierung unter MS-DOS 67
3.1 Grundlagen Assembler . . . 68
3.1.1 Die INTEL Prozessorfamilie . . . 68
3.1.2 Das Segment-Konzept . . . 69
3.1.3 Adressierung (Besonderheiten) . . . 74
3.1.4 Interrupts . . . 79
3.2 Grundlagen Hochsprachen . . . 81
3.2.1 Speichermodelle . . . 81
3.2.2 Lokale Variablen . . . 82
3.2.3 Parameterübergabe . . . 83
3.2.4 Der Bindeprozeß . . . 88
3.3 Der Aufbau von MS-DOS . . . 94
3.3.1 Kommandoebene . . . 94
3.3.2 DOS-Kernel (Interruptebene I) . . . 96
3.3.3 Gerätetreiber . . . 97
3.3.4 BIOS (Interruptebene II) . . . 99
3.4 Verwaltung interner Speicher . . . 100
3.4.1 Organisation des Arbeitsspeichers . . . 101
3.4.2 Start und Struktur von Programmen . . . 104
3.5 Verwaltung externer Speicher . . . 111
3.5.1 Master- und Partition-Boot-Record . . . 111
3.5.2 Funktionen zur Dateibearbeitung . . . 116
3.5.3 Der Urladevorgang . . . 120
3.6 Ein- und Ausgabe . . . 122
3.7 Speicherresidente Programme . . . 123
3.7.1 TSR-Programme . . . 123

3.7.2 Gerätetreiber 124
3.7.3 Vor- und Nachteile 125

4 Entwicklung der Systemprogramme 127
4.1 Anforderungsanalyse 128
4.1.1 Kommandoebene 132
4.1.2 Interruptebene 133
4.1.3 Der Urladevorgang 137
4.1.4 Qualität der Schutzmaßnahmen 137
4.2 Systementwurf 138
4.2.1 Bibliotheken 138
4.2.2 Prüfprogramme 140
4.2.3 Neue Kommandos 141
4.2.4 Residente Programme 142
4.2.5 Hilfsprogramme 144
4.3 Prüfprogramme 145
4.3.1 Check System (ChkSys) 145
4.3.2 Seal und Check Seal (chk_seal) 152
4.3.3 Check State (ChkState) 158
4.4 Kommandos mit Kontrollfunktionen 171
4.4.1 AVCopy 171
4.4.2 AVRename 180
4.4.3 Deaktivierung interner Kommandos 186
4.5 Realisierung des residenten Teils 186
4.5.1 Das Interrupt/“C”-Interface Std_INTC 186
4.5.2 Die TSR-Plattform Std_TSR 194
4.5.3 TSR-Programme und Datenzugriff 204
4.5.4 AVWatch 206
4.5.5 AVWatchG(lobal) 207
4.5.6 AVWatchI(ntegrity) 208
4.5.7 AVWatchP(artition) 210
4.5.8 AVWatchF(ile) 217
4.5.9 Mögliche Erweiterungen 229
4.5.10 AVConfig 231

5 Diskussion 239
5.1 Einschränkungen 239
5.1.1 Abwehr 239
5.1.2 Selbstschutz 243
5.2 Ausblick 244
5.2.1 Zukünftige Entwicklung 244
5.2.2 Sinnvolle Erweiterungen 245
5.2.3 “Computerviren, das Universum und der ganze Rest” 245

A Software 253
A.1 Die Begleitdiskette 253

A.2 Software zur Programmerstellung 256
A.3 MSDOS_S.LIB 258
A.3.1 Kernel-Interrupts 258
A.3.2 Video-Interrupt (BIOS) 273
A.3.3 Disk-Interrupt (BIOS) 277
A.3.4 Keyboard-Interrupt (BIOS) 277
A.4 AVSys.LIB 279
A.4.1 Allgemein einsetzbare Funktionen 279
A.4.2 Bearbeitung von Dateinamen 285
A.4.3 Stringverarbeitung 289
A.4.4 Listenverwaltung 291
A.4.5 Zugriff auf die Kommandozeile 292
A.4.6 Spezielle Funktionen 295

B Öffentliche Datennetze **301**
B.1 Grundlagen Electronic Mail 302
B.2 Besonderheiten (UUENCODE) 303
B.3 LISTSERV-Server (Diskussionslisten) 305
B.4 TRICKLE-Server 306
B.5 FTP-Server 307
B.6 NETSERV-Server 308
B.7 Mailboxen 308
B.8 “Der Netz-Knigge” 309

C Informationen zu MS-DOS **311**
C.1 Kommandos, Interrupts, Funktionen 311
C.2 ASSIGN, JOIN, SUBST 322

D Abkürzungsverzeichnis **325**

E Glossar **327**

Einführung

Was müssen Programme zur Abwehr von Computerviren und anderen Softwareanomalien leisten? Wie werden sie konzipiert und implementiert? Im Laufe von fünf Kapiteln werden diese und andere Fragen im Hinblick auf den IBM-PC unter MS-DOS beantwortet.

Die in diesem Buch entwickelten Programme sind in der Programmiersprache "C" und in 80*86 Assembler geschrieben, wobei der Anteil an Assembler möglichst gering gehalten wurde. Grundkenntnisse in "C" werden vorausgesetzt, Programmierfertigkeiten in Assembler sind nicht notwendig. Die nötigen Informationen für die Systemprogrammierung unter MS-DOS vermittelt das Buch. Alles, was für das Verständnis der Programme und der Feinheiten der INTEL-Prozessorfamilie notwendig ist, wird ausführlich und anhand von Beispielen erläutert.

Das Kapitel "Theorie der Softwareanomalien" definiert den Begriff Softwareanomalie sowie eine Reihe weiterer Termini aus diesem Umfeld. Jeder Anomalietyp wird beschrieben und anhand eines Fallbeispiels aus der Praxis in Funktion und Auswirkungen anschaulich vorgestellt. Eine kleine Historie der Computerviren stellt die Experimente von Fred Cohen und die Entwicklung der Computerviren bis heute vor. Eine Übersicht über die Rechtslage und Maßnahmen öffentlicher Stellen beschließt das Kapitel.

Im Kapitel "Theorie der Abwehr" werden aus Informationen zur Funktion von Softwareanomalien Strategien zu ihrer Abwehr entwickelt. Betrachtet werden sowohl technische als auch menschliche Aspekte. Welche Rolle spielen die Anwender bei der Verbreitung und Verseuchung? Inwiefern eignen sich konventionelle Sicherheitsmaßnahmen zur Abwehr von Softwareanomalien? Ein Vergleich von MS-DOS mit anderen Betriebssystemen zeigt generelle Sicherheitslücken auf. Cohen's Theorien zur Abwehr und kommerziell verfügbare Schutzkonzepte werden anhand von Fallbeispielen vorgestellt und kritisch auf ihre Schutzwirkung hin untersucht. Den Abschluß bildet die Entwicklung und Diskussion alternativer Methoden auch aus ungewöhnlichen Ansätzen (z.B. Analogien zur Biologie) heraus.

Das Kapitel "Systemprogrammierung unter MS-DOS" vermittelt die nötigen Kenntnisse zur Entwicklung von Systemprogrammen gegen Computerviren. Eine Einführung in die Architektur der INTEL-Prozessoren, 80*86-Assembler und die Zusammenarbeit zwischen Maschinensprache und Hochsprachen wie "C" eröffnet das Kapitel. Allgemeine Grundlagen zum Aufbau von MS-DOS, Realisierung von Diensten und der Verwaltung von internem und externem Speicher legen den Grundstein für komplexere Anwendungen wie TSR-Programme und Interrupt-Handler.

Das Kapitel "Entwicklung der Systemprogramme" vereinigt die erarbeiteten Abwehrkonzepte und Kenntnisse der Systemprogrammierung miteinander. Für den Leser mitverfolgbar und selbständig zu erweitern werden Schutzprogramme für PCs unter MS-DOS entwickelt, die sich für den Betrieb "zu Hause" und in Rechenzentren (z.B. Hochschulen) eignen.

Den Abschluß bildet das Kapitel "Diskussion". Zu den entwickelten Programmen wird kritisch Stellung bezogen. Wo und worin liegen Schwächen in Abwehr und Selbst-

schutz? Ist Schutz überhaupt möglich? Wie sieht die Zukunft aus? Was könnte man sinnvollerweise noch verbessern und erweitern? Auch Kurioses, Merkwürdiges, Erstaunliches und Befremdliches kommt nicht zu kurz. Es geht um Dinge wie die militärische Anwendung von Computerviren und die Frage "Können Viren spontan entstehen?".

Die Anhänge bieten noch einmal reichlich Information zum Thema. Anhang A beschreibt den Inhalt und die Arbeit mit der Begleitdiskette, die separat bestellt werden kann (s. beigefügte Bestellkarte). Eine ausführliche Beschreibung der beiden Systembibliotheken erläutert Funktionen, die aus Gründen der Übersichtlichkeit aus dem Haupttext ausgelagert wurden.

Informiert sein ist gut, informiert bleiben ist wichtig. Deshalb enthält der Anhang B eine Anleitung zur Benutzung von Diskussionslisten, Server-Diensten und Mailboxen. Diese ermöglicht jedem mit Zugriff auf das DFN, EARN, BITNET oder INTERNET den Einstieg in die aktuellste und umfangreichste Informationsquelle, die es für Softwareanomalien und viele andere Themengebiete aus dem Computersektor gibt.

Der Anhang C enthält Informationen zu MS-DOS. Dort finden sich Tabellen der MS-DOS- und OS/2-Kommandos, der Interrupts sowie der Kernel- und BIOS-Funktionen. Dazu kommt eine Beschreibung der internen Arbeitsweise der Kommandos `assign`, `join` und `subst`. Abkürzungsverzeichnis, Glossar und Index beschließen das Buch.

Kapitel 1

Theorie der Softwareanomalien

Eine Anomalie (griech.) ist eine Ausnahme von der Regel, eine Abweichung vom Normalfall. Unter einer Software-Anomalie ließe sich demnach eine Abweichung des tatsächlichen vom erwarteten Verhalten eines Programms verstehen. Die Ursachen hierfür sind im klassischen Fall Fehler in der theoretischen Konzeption oder Programmierfehler, die bei der Umsetzung der entwickelten Algorithmen in die Zielsprache gemacht wurden. Falls diese beiden Fehlerquellen ausgeschlossen werden können, bleibt als Grund für abweichendes Verhalten nur der Programmcode übrig, der nicht zur Realisierung der geplanten und vom Anwender aus der Beschreibung des Programms erwarteten Funktionen dient. Diese Programmteile wurden entweder bei der Produktion der Software absichtlich implementiert (Trojaner) oder kamen erst später zum fertigen, ausführbaren Programm hinzu (Viren). Weitet man den Begriff "erwartetes Programmverhalten" auf "allgemein erwartetes Wohlverhalten eines Programms" aus, fallen auch die Würmer, deren Zweck nur in ihrer Verbreitung besteht, in die Kategorie der Softwareanomalien.

Def. Softwareanomalie: Eine Softwareanomalie ist eine Abweichung des Programmverhaltens vom erwarteten Verhalten, die ihre Ursache weder in konzeptionellen noch programmiertechnischen Fehlern hat.

Der folgende Text definiert die bereits angesprochenen drei verschiedenen Typen von Softwareanomalien, beschreibt schematisch ihre Funktion und erläutert diese und für das Verständnis der folgenden Kapitel notwendige Fachbegriffe anhand von Fallbeispielen aus der Praxis.

1.1 Würmer

1.1.1 Definition

Def. Wurm: Ein Wurm ist ein eigenständiges Programm, das Kopien seiner selbst erzeugt und zum Ablauf bringt.

Etymologie. Worauf sich die Bezeichnung "Wurm" bezieht, ist etwas unklar; vermutlich rührt der Name von der Eigenschaft dieser Programme her, von System zu System zu "kriechen" und sich von Benutzer zu Benutzer "durchzufressen". Dabei befinden sich "Segmente" des Wurms gleichzeitig in verschiedenen Systemen. Die Bezeichnung "Bakterie" oder "Amöbe" wäre von der Komplexität her angemessener, wenn ähnlich wie bei "Virus" eine Parallele zu einfachen biologischen Lebensformen gezogen werden soll. Sowohl Bakterien als auch Amöben sind allein lebensfähig und können sich unter geeigneten Bedingungen durch Teilung vermehren. Da aber "Wurm" die eingebürgerte Bezeichnung ist, wird diese hier auch im weiteren verwendet.

Wurmprogramme werden von ihrem Erzeuger an einer beliebigen Stelle ins Netz gebracht und gestartet. Die weitere Verbreitung erfolgt dann automatisch ohne weiteres menschliches Zutun. Ein Wurm pflanzt sich fort, indem er sich selbst dupliziert und die Kopie startet. Das erzeugende Programm kann nur dann weiter aktiv bleiben, wenn es entweder möglich ist, mehrere Programme auf einem Rechner gleichzeitig ablaufen zu lassen (Multitasking) oder andere Rechner, z.B. über Netzwerke, zu nutzen. Da auf einem isolierten Rechner keine flächige Verbreitung erfolgen kann, sind Wurmprogramme typisch für Netzwerke, über die die Duplikate automatisch verschickt, und, ein wichtiger Punkt, aktiviert werden. Der quasi ferngesteuerte Start ist ein kritischer Aspekt bei der Wurm-Programmierung, weil die meisten Netzwerkprogramme diese Möglichkeit nicht vorsehen oder aus Sicherheitsgründen verbieten. Das es trotzdem geht, zeigt das folgende Beispiel aus der Praxis, in dem ein Wurmprogramm Sicherheitslücken unter UNIX clever und gezielt ausnutzt.

1.1.2 Trapdoors, triviale Paßwörter, Bugs: Der INTERNET-Worm

Am 2. November 1989 startete Robert Tappan Morris, 23, Student an der amerikanischen Cornell-Universität, ein Wurm-Programm, das unter dem Namen INTERNET-Worm weltweite Berühmtheit erlangte [66, 67]. Das INTERNET ist ein komplexes Netzwerk unterschiedlichster Rechner, das sich über Jahre hinweg entwickelt hat und vor allen Dingen Universitäten und Forschungsstätten miteinander verbindet. Die Vorgehensweise des Wurmprogramms zeigt gleich drei Arten von Schwachstellen auf, die ganz allgemein Angriffspunkte für Rechnerattacken bilden und die z.T. nur schwer zu vermeiden oder zu beheben sind.

Trapdoors. Durch eine sog. *Hintertür* im `sendmail`-Programm der attackierten UNIX-Rechner war es möglich, ein Programm nicht nur zu verschicken, sondern auf dem Zielrechner auch ausführen zu lassen. Eine Hintertür ist eine vom Programmierer

eingebaute Funktion, mit deren Hilfe sich dieser nach Installation des Programms immer wieder Zugriff auf sonst nicht verfügbare, durch Sicherheitsfunktionen geschützte Dienste verschaffen kann. Im klassischen Fall handelt es sich dabei um ein besonderes Paßwort, das unbefugten Zugang zum System ermöglicht ("Joshua" in "Wargames", US-Spielfilm) oder auch, wie beim `sendmail`-Programm, um eine Funktion zur Ermöglichung der Fernwartung. Abhilfe wäre nach dem "4-Augen-Prinzip" möglich, nach dem bei einem sicherheitskritischen Vorgang mindestens zwei Personen anwesend sein müssen, um Manipulationen durch einen Einzeltäter zu verhindern. Im konkreten Fall müßte der Quelltext von einer oder mehreren Personen kontrolliert werden, die am besten nichts mit der Programmierung selbst zu tun haben.

Triviale Paßwörter. Einmal im Zielsystem aktiviert, versuchte der Wurm mit Hilfe einer mitgeführten Liste gebräuchlicher Paßwörter und Permutationen der vorgefundenen Benutzernamen Zugriff auf den Rechner zu erhalten. Die dahinter steckende Überlegung war, daß viele Anwender triviale Paßwörter benutzen, zu denen z.B. der eigene Name, der Name des Freundes/der Freundin, die Initialen, die Benutzerkennung, Automarken und bekannte Persönlichkeiten gehören [98]. Wenn der Einbruchsversuch glückte, und das war leider oft genug der Fall, verschickte sich das Programm erneut an den nächsten Rechner und breitete sich so auf dem gesamten Netzwerk aus.

Bugs. Zudem nutzte Morris einen *Bug* (am. sl. für Fehler) im Systemprogramm `fingerd` aus, das eine Liste der aktuellen (eingeloggten) Benutzer des System ausgibt. Dieses Programm erwartet beim Aufruf eine Reihe von Parametern, von denen einige intern in Arrays (Feldvariablen) gespeichert werden. Der Fehler in `fingerd` bewirkte nun, daß bei einem bestimmten, zu langen Argument ein Array überlief und die dahinter liegenden Speicherzellen überschrieb[1]. Auf diese Weise konnten interne, normalerweise nicht zugängliche Daten und Teile des Programmcodes verändert werden. Da `fingerd` mit den Privilegien eines Systemadministrators abläuft, war es möglich, bei geschickter Wahl des Parameters Zugriff auf Systemfunktionen zu erhalten. Wegen der fehlenden Gewaltenteilung unter UNIX bedeutete dies, daß dem Wurmprogramm alle Funktionen uneingeschränkt zur Verfügung standen, die das Betriebssystem überhaupt anbietet.

An dieser Stelle drängt sich die Frage auf, ob der Quelltext von Betriebssystemen, wie im Falle von UNIX, veröffentlicht werden sollte oder nicht. Neben dem Vorteil, selbst am Quellcode Anpassungen und Erweiterungen vornehmen zu können, ergibt sich für entsprechend Interessierte die Möglichkeit, gezielt nach Schwächen der Programmierung Ausschau zu halten und diese für eigene Zwecke zu nutzen. Bliebe der Quelltext dagegen geheim, würden Einbrüche über Software-Bugs wie z.B. im `fingerd`-Kommando sehr erschwert, wenn nicht unmöglich. Die Antwort auf die oben gestellte Frage hat einen eher philosophischen Charakter. Das Wissen um einen Sachverhalt ist nicht erst seit der Erforschung der Kernenergie meist ambivalent zum Wohle und zum Schaden anderer verwendbar. Auch die in diesem Buch vermittelten Kenntnisse lassen sich neben der Abwehr auch zur Programmierung von Computerviren einsetzen. Dennoch ist der Autor der Meinung, daß der freie Austausch von Information der Isolation vorzuziehen ist, weil Wissen nur so erweitert und gewonnen werden kann.

Daß sich der INTERNET-Worm so spektakulär entwickelte, lag an einer Fehlein-

[1] Die Sprache "C", in der `fingerd` geschrieben ist, fängt solche Fehler nicht automatisch ab.

schätzung des Programmierers. Das Programm befiel, um unnötiges Aufsehen durch Verbrauch an Rechenzeit zu vermeiden, nur Rechner, die noch nicht von der Infektion erreicht worden waren. Da sich durch Vortäuschung einer Verseuchung (Immunisierung) die Möglichkeit ergeben hätte, den Wurm zu stoppen, sah Morris vor, daß das Programm mit einer Wahrscheinlichkeit von $p = 0.1$ auch bereits infizierte Rechner befallen sollte. Angesichts der hochgradigen Vernetzung der Rechner erwies sich dieser Wert als viel zu groß gewählt. Die Rechner, die Opfer einer vielfachen Infektion wurden, brachen unter der zusätzlichen Rechenlast zusammen. Zum Glück für die Rechnerbetreiber machte Morris von seinen Möglichkeiten (s.o., "Bugs") keinen Gebrauch, sondern beschränkte sich darauf, Paßwörter auszuspähen.

Ein anderer bekannter Fall ist der des DECNET- oder WANK[2]"-Worms, der Teile des NASA-eigenen Computernetzes befiel.

1.2 Trojaner

1.2.1 Definition

Def. Trojaner: Ein Trojaner ist ein eigenständiges Programm, das unter anderem Funktionen ausführt, die nicht Bestandteil der vorgeblichen Programmfunktion sind.

Etymologie. Der Name rührt vom berühmten Trojanischen Pferd her, mit dessen Hilfe die Griechen einst Troja eroberten. Nach zehnjähriger erfolgloser Belagerung der stark befestigten Stadt verfielen die Griechen auf eine List. Sie bauten ein großes hölzernes Pferd, in dessen Inneren sich Soldaten versteckt hielten, und zogen sich zum Schein zurück. Die Trojaner hielten das Pferd für ein Geschenk der Götter und transportierten es in ihre Stadt. Die Folgen für Troja und seine Bevölkerung sind allgemein bekannt. Auch wenn Programme heute oft keine Göttergeschenke sind, enthalten doch manche gefährliche Zusatzfunktionen, die nicht zum erwarteten Programmverhalten gehören und von denen der Anwender bis zur Zeitpunkt der unerfreulichen Überraschung nichts ahnt. Im Folgenden dazu ein Beispiel.

1.2.2 Mangelnde Vorsicht: Der AIDS-Trojaner

Anfang Dezember 1989 verschickte die Firma PC Cyborg Corporation mit Sitz in Panama von London aus 20000 (geschätzt) Kopien des PC-Programms "AIDS-Information" an Institute und Privatleute in Deutschland, England und Skandinavien [63, 64]. Benutzt wurde offensichtlich der Verteiler der PC-Business World und die Teilnehmerliste einer internationalen AIDS-Konferenz.

Mangelnde Vorsicht. Ein Aufkleber auf der Diskette erklärt in vier Schritten knapp und einfach die Installation des Programms auf der Festplatte. Die meisten der

[2]Worms Against Nuclear Killers

später von den Folgen des Trojaners betroffenen Benutzer ließen es bei dieser Lektüre bewenden und starteten das Programm. Diejenigen, die — mit Recht — größere Vorsicht walten ließen, studierten zunächst die begleitende Broschüre, die jeden aufmerksamen Anwender hätte stutzig machen müssen. Sie enthält Informationen zum Programm, dessen Bedienung und als größten Teil einen Abschnitt über eine "Limited Warranty" (eingeschränkte Garantie) und ein "License Agreement" (Nutzungsabkommen). In letzterem wird die Einsendung von US$189 bzw. US$378 gefordert und für den Fall der Nichtzahlung der Einsatz von Mitteln angekündigt, "die sicherstellen, daß der Benutzer das Programm nicht mehr benutzen kann" und "die andere Programme nachteilig beeinflussen".

Das Informations-Programm selbst ist laut Aussagen von Medizinern wertlos. Viel aufwendiger programmiert und weitreichender in seinen Folgen ist das, was der Benutzer nicht erwartet hat. Nach einer gewissen Anzahl von Systemstarts wird eine Funktion aktiv, welche die Drohungen in Wirklichkeit umsetzt und den Inhalt der Festplatte verschlüsselt. Ende 1990 schließlich wurde Dr. Joseph W. Popp als Tatverdächtiger festgenommen [65].

1.3 Computerviren

1.3.1 Definition

Def. Computervirus (kurz: Virus): Ein Virus ist ein in einem Wirtsprogramm enthaltener Programmteil, der Kopien seiner selbst erzeugt, in dem er andere Programme verändert.

Etymologie. Ein Virus (von lat. virus = giftiger Saft; in der Medizin sächlich, sonst auch maskulin; Plural Viren) ist in der Biologie ein Zwitter zwischen großem Eiweißmolekül und Lebensform. Viren können in Kristallform extreme Umweltbedingungen überstehen und haben dann die Eigenschaften toter Materie. Sie sind allein weder lebensfähig, noch können sie sich vermehren. Dazu benötigt das Virus eine Wirtszelle, in die es seine Erbinformation einschleust und so zur Produktion von Virenduplikaten zwingt. Die Zelle stirbt bei der Freisetzung der Viren ab.

In der einschlägigen Literatur zu Computerviren haben sich bereits einige, zumeist englische Fachbegriffe, eingebürgert, die auch in diesem Buch Verwendung finden. Die folgenden Abschnitte definieren und erläutern diese Termini anhand der im Text durchgespielten Infektion eines *sauberen*, d.h. vormals virenfreien Rechners. Eine gute Einführung in die Computervirenproblematik geben die Bücher [29] (sehr umfassend) und [26] (komprimiert) sowie die Broschüre [32] mit knappen Tips für die Praxis.

1.3.2 Transport (Vektor)

In der Biologie bezeichnet man das Medium, das einen Krankheitserreger transportiert, als *Vektor* einer Infektion. Dazu kommen alle Lebewesen und Gegenstände in Frage, in

oder auf denen der Keim überleben kann. Was bei der Pest des Mittelalters Ratten und Flöhe bewerkstelligten, wird im Zeitalter der Computer und der Computerviren von Menschen, die austauschbare Datenträger wie Disketten und Wechselplatten transportieren, oder von der Datenfernübertragung über Kabel und Funk übernommen. Alle Medien, die Programmcode speichern oder übermitteln können, sind potentielle Vektoren eines Computervirus.

Zu unserem Fallbeispiel. Auf einem Rechner befinde sich auf Floppy Disk oder Festplatte ein Programm, das einen Virus enthält. Ein solches Programm nennt man *infiziert* oder *verseucht*. Der Begriff "Programm" ist weiter als gemeinhin gefaßt und bezieht sich auf eine Datei, die

- direkt vom Betriebssystem geladen und ausgeführt werden kann (ausführbare Datei; engl. executable file). Dazu zählt auch Code, der sich z.B. unter MS-DOS im Bootsektor von Disketten und Festplatten befindet und beim Start des Computers automatisch ausgeführt wird.
- durch Interpreter interpretiert oder durch Compiler in eine ausführbare Datei übersetzt werden kann (Quelltext; engl. source code) oder
- ein Zwischenprodukt eines Interpretations- oder Kompilationsvorgangs darstellt (Objektdateien, Bibliotheken)

Da jedwede von anderen Programmen interpretierte Daten im Prinzip ein Programm darstellen, ergeben sich für Softwareanomalien breite Ansatzmöglichkeiten. Dazu ein paar exemplarische Beispiele:

- Ist unter MS-DOS der ANSI-Gerätetreiber für den Bildschirm geladen (`ansi.sys` o.ä.), interpretiert dieser Steuersequenzen, die in der Textausgabe auf den Bildschirm enthalten sind. Ein mit dem `type`-Kommando ausgegebener "trojanischer Text" könnte auf diese Weise die `Return` -Taste mit dem Text "`DEL *.exe` `Return`" belegen...
- Ein in POSTSCRIPT (Sprache zur Druckersteuerung) geschriebener Trojaner könnte das Paßwort des Druckers so umdefinieren, daß dieser vom Rechner nicht mehr angesprochen werden kann.
- Viele Kalkulationsprogramme sind vom Funktionsumfang her mächtig genug, um die Programmierung eines in der jeweiligen Makrosprache geschriebenen Virus oder Trojaners zu ermöglichen.

Einschränkend und zugleich beruhigend ist anzumerken, daß nicht jedes Programm, das Daten interpretiert, auch mächtig und allgemein genug ist, um die Funktionen anzubieten, die ein Computervirus zu seiner Verbreitung benötigt. Trotzdem sollen die angeführten Beispiele das Gespür für mögliche Gefahrenquellen wecken.

1.3.3 Aktivierung und Residenz

Die *Aktivierung* des Virus erfolgt durch den Start des Wirtsprogramms. Im Laufe der Ausführung oder Interpretation wird der Viruscode irgendwann, meist vor dem Start

des Originalprogramms, erreicht. Ein Computervirus zwingt dadurch seinem Wirtsprogramm trojanische Eigenschaften auf, denn die Ausführung des Viruscodes gehört nicht zur ursprünglich beabsichtigten Programmfunktion. Insofern handelt es sich bei einem Virus um einen sich automatisch verbreitenden Trojaner. Die Vorgehensweise eines Virus bei Infektion und Manipulation läßt sich zwecks besserer Übersicht in verschiedene Klassen einteilen, die im Folgenden besprochen werden.

Direct Action. Ein Direct Action-Virus sucht nach seiner Aktivierung sofort nach anderen, noch nicht befallenen Programmen, infiziert diese und startet dann das Wirtsprogramm. Das Virus wird — aus Tarnungsgründen — nur kurz aktiv, führt die ihm zugedachten Aufgaben aus und beendet sich, indem es die Kontrolle vollständig abgibt. Bis zum nächsten Aufruf eines verseuchten Programms wird kein Viruscode mehr ausgeführt (s.Abb. 1.1, oberes Diagramm). Diese Infektionsmethode hat den Vorteil, daß nur normale Betriebssystemfunktionen und Rechte erforderlich sind; spezielle Manipulationen auf Systemebene sind nicht notwendig. Solche Viren sind auf jedem System sehr einfach zu implementieren und kommen mit einem Minimum an Programmbefehlen aus, wie ein nur 42 (zweiundvierzig) Bytes großes Virus unter MS-DOS bewiesen hat.

Indirect Action. Das Virus oder ein Teil davon installiert sich nach seiner Aktivierung resident im Speicher, gibt die Kontrolle ebenfalls ab, bleibt aber während das Wirtsprogramm abläuft und darüber hinaus präsent und wird durch bestimmte Ereignisse aktiviert. Dies geschieht, indem das Virus für sich Speicher reserviert und Interruptvektoren auf Routinen des Virusprogramms umsetzt[3]. Dazu sind Eingriffe auf Betriebssystemebene notwendig, zu der normale Anwender auf Rechnern mit Schutzeinrichtungen keinen Zugang haben. Auf PCs und Homecomputern dagegen sind solche Manipulationen der Regelfall, weil deren Betriebssysteme über keinerlei Schutzmechanismen verfügen. Dazu ein Beispiel: In vielen Systemen existiert ein Interrupt, der von der Hardware periodisch ausgelöst wird und ebenso oft eine Serviceroutine aufruft (unter MS-DOS ungefähr alle 54.92 ms = $\frac{1}{18.21}$ sec). Ein Virus, das diesen Interrupt abfängt, wird ebensooft aktiviert und hat die Möglichkeit, Operationen auszuführen (s.Abb. 1.1, unteres Diagramm).

Infect on Execute/Open. Speicherresidente Viren haben viel mehr Handlungsmöglichkeiten als gewöhnliche Viren, da sie über einen längeren Zeitraum präsent sind und ereignisgesteuert auf Systemzustände reagieren können. Ein Direct Action-Virus kann nur während einer, wegen der angestrebten Unauffälligkeit möglichst kurzen Zeitspanne nach dem Start andere Programme infizieren (→ Infect on Execute). Darüber hinaus muß es sich seine Opfer selbst aktiv suchen; eine weitere, durch zusätzliche Laufwerkszugriffe potentiell auffällige Aktion. Ein Indirect Action-Virus kann sich z.B. in Betriebssystem-Funktionen einklinken, die zum Öffnen (→ Infect on Open), Lesen, Schreiben von Dateien und Starten von Programmen dienen. Beim Aufruf einer dieser Funktionen durch ein anderes Programm bekommt das Virus die Namen infizierbarer Programme praktisch vom Betriebssystem selbst angeliefert; der anschließende Infektionsvorgang ist durch die sowieso angeforderte Laufwerksaktivität gut getarnt.

[3]Dazu mehr unter 3.1.4 “Interrupts”.

Abbildung 1.1: Aktivierung und Residenz von Viren

Single/Multiple Infector. Diese Unterscheidung erfolgt je nachdem, ob ein Virus pro Infektionsvorgang nur ein oder gleich mehrere Programme infiziert. Die meisten Viren sind vom Single Infector-Typ, weil die Infektion mehrerer Programme hintereinander evtl. vom Benutzer wahrnehmbare Verzögerungen und Diskettenoperationen verursacht.

Infektions-/Manipulations-Trigger. Viele, besonders die moderneren "Sophisticated" (engl.: raffiniert) Viruses, beschränken sich aus den unterschiedlichsten Gründen in ihrer Verbreitungs- und Manipulationstätigkeit. Mögliche Motive sind

- möglichst weite Verbreitung
  - langsame Verbreitung und unauffällige, subtile Manipulationen oder späte Auslösung der Schadensfunktion (→ schleichende Infektion)
  - schnelle Verbreitung, sofortiger Schaden
- Vortäuschen eines möglichst schwer lokalisierbaren und identifizierbaren Fehlers (sporadische, simulierte Tippfehler; Fehler bei Datenübertragung zu und von Peripheriegeräten etc.)
- Schadensauslösung an einem bestimmten Datum (z.B. Jahrestage politischer Ereignisse)

Verschiedene, gleichzeitig vorhandene Zustände aktivieren einen Trigger (engl.: Abzug, Auslöser), der die Verbreitungs- oder die Manipulationsfunktion auslöst. Häufig werden Systemdatum und -zeit, Dateidatum, -zeit, -länge und -inhalt sowie bestimmte Ereignisse wie Interrupts und Tastatureingaben für diesen Zweck herangezogen.

1.3.4 Infektionsmechanismus (Typologie)

Überschreibende Viren. Die programmtechnisch einfachste Virenform ist die der überschreibenden Viren. Der Viruscode wird dabei einfach über den Anfang des zu infizierenden Programms kopiert, das dadurch zerstört wird (s. Abb. 1.2, "Overwrite"). Deshalb ist eine Infektion mit überschreibenden Viren leicht zu entdecken, denn infizierte Programme funktionieren nicht mehr korrekt oder stürzen vollständig ab. Bei dieser Methode bleibt das Wirtsprogramm in der Länge unverändert, was ein Vorteil bei der Tarnung ist.

Besondere überschreibende Viren. Eine besonderes Exemplar und Spezialfall der überschreibenden Viren ist das Lehigh-Virus, das ausschließlich den MS-DOS-Kommandointerpreter befällt. Das Virus benutzt dabei einen reservierten, aber leeren Datenbereich in `command.com` und kann sich so, ohne die Länge des Programms zu verändern oder dessen Funktion zu beeinträchtigen, im Programmcode einnisten (s.Abb. 1.2, "Lehigh-Typ"). Da solche Datenbereiche nicht in jedem Programm vorkommen oder zweifelsfrei identifizierbar sind, ist das Lehigh-Virus auf `command.com` fixiert und kann keine anderen Programme befallen.

Es existiert eine weitere Möglichkeit, ungenutzten Speicherplatz zur Speicherung von Viruscode zu verwenden, nur diesmal nicht im Inneren des Wirtsprogramms, sondern in ungenutzten Bereichen, die durch die Methode der Dateiorganisation entstehen. Datenträger wie Disketten und Festplatten werden in Untereinheiten von Speicherblöcken (engl. blocks) verwaltet, die entweder frei oder belegt sind. Da Dateien meist nicht an der letzten Adresse des letzen Blocks enden, bleibt dort ungenutzter, für den Anwender scheinbar nicht vorhandener Speicher frei.

Dazu ein Beispiel: die Blockgröße betrage 4096 Bytes (4 kB), das Programm habe eine Länge von 8193 Bytes. Die Programmdatei belegt demnach drei Blöcke, von denen die ersten zwei vollständig genutzt werden ($2 * 4096 = 8192$ Bytes), der letzte aber nur mit einem Byte belegt ist ($8193 - 2 * 4096 = 1$ Byte). Die anderen 4093 Bytes sind für die Speicherung von Dateien verloren, aber prinzipiell frei verwendbar.

Link-Viren. Die nächste Entwicklungsstufe der Virenprogrammierung ist die der *Link-Viren*, die sich an das Wirtsprogramm in nicht zerstörerischer Weise anhängen (engl. to link = verketten, verbinden) und so dessen Funktionsfähigkeit erhalten. Diese lassen sich noch einmal, je nach dem, ob sich das Virus dem Programmcode voranstellt oder sich an ihn anhängt, in zwei Unterklassen gruppieren, nämlich die der *Prepend*- und die der *Append*-Viren. "Prepend" ist ein von Cohen eingeführtes Kunstwort für "voranhängen", welches das Vorgehen dieses Virentyps treffend beschreibt.

Prepend-Viren. Viren, die sich Programmen voranstellen, werden dadurch bei Programmstart automatisch zuerst ausgeführt (s. Abb. 1.2, "Prepend"). Dabei gibt es allerdings drei Einschränkungen für die Programmstruktur des Wirtsprogramms, die vom betrachteten Betriebssystem, genauer: dessen Programm-Lader abhängen:

1. Die Ausführung muß tatsächlich am Programmanfang beginnen. Dies ist z.B. bei `*.exe`-Programmen unter MS-DOS nicht der Fall, bei denen die Startadresse

in einem besonderen Programmvorspann, dem `exe`-Header, festgelegt ist. Der Viruscode wird deshalb nicht gestartet.

2. Das Programm darf nur relative Adressierung verwenden, weil es durch das Virus um dessen Länge im Adreßraum nach hinten verschoben wird. Adressierung — selbst relativ zum Programmanfang — wird dadurch fehlerhaft.
3. Programme mit einem besondererem Vorspann, z.B. zur Berechnung von Adreßbezügen (wie in MS-DOS-`*.exe`-Dateien), werden durch Prepend-Viren unbrauchbar, weil der Lader den Viruscode als Vorspanndaten interpretieren würde.

Durch die gestellten Bedingungen bleiben nur Programme übrig, deren Ausführung beim ersten Byte der Datei beginnt und die nur relative Adressierung verwenden (z.B. `*.com`-Programme unter MS-DOS). Ein weiterer Nachteil besteht darin, daß vom Betriebssystem her Dateien prinzipiell immer nur am Ende erweitert werden können. Das bedeutet für das Prepend-Virus, daß es das gesamte Wirtsprogramm zeitraubend und daher auffällig umkopieren muß. Wahrscheinlich deshalb und wegen der oben angeführten Schwierigkeiten sind kaum Viren dieses Typs bekannt (z.B. “Fu Manchu”).

Abbildung 1.2: Infektions-Typologie

Append-Viren. Dieser Virus-Typ hängt sich, wie schon aus dem Namen hervorgeht, an das Wirtsprogramm hinten an (s. Abb. 1.2, “Append”). Da alle Betriebssysteme das Erweitern von Dateien am Ende unterstützen und der Viruscode meist nur eine rel. geringe Anzahl von Bytes umfaßt, ist diese Infektionsmethode schnell und einfach zu implementieren. Im Unterschied zu Prepend-Viren haben Append-Viren den Nachteil, daß an das Programmende angefügter Code ohne zusätzliche Maßnahmen nie zur Ausführung gelangt. Um dennoch beim Programmstart aktiviert zu werden, wird

die Startsequenz des Originalprogramms so verändert, daß zuerst der Viruscode zum Zuge kommt. Hat das Virus seine Aufgabe ausgeführt, restauriert es den manipulierten Teil des Wirtsprogramms und startet es von neuem.

Link-Viren erhalten zwar die Funktion des Wirtsprogramms, bewirken aber eine Vergrößerung der Datei, die einem aufmerksamen Anwender oder einem Prüfprogramm auffallen könnte. Der Abschnitt 1.3.5 "Tarnung" befaßt sich mit Viren, die (erfolgreich) versuchen, den Anwender und sogar Prüfprogramme zu täuschen.

Bootviren. Beim Start eines IBM-PCs wird zunächst das Betriebssystem von Diskette oder Festplatte geladen. Im Laufe des Bootprozesses wird im sog. Boot- und Partitionsektor befindlicher Programmcode automatisch geladen und ausgeführt. Der Abschnitt 3.5.3 "Der Urladevorgang" beleuchtet das Verfahren noch im Detail. Bootviren (oder BSIs; **B**oot **S**ector **I**nfector) nutzen diesen Umstand für ihre Zwecke aus und werden dadurch zu einem Zeitpunkt aktiviert, der vor dem Start jedes Schutzprogramms liegt und zu dem der Anwender keinerlei Einflußmöglichkeiten hat; ein unschätzbarer Vorteil für die "böse" Seite. Gegen die Infektiosität von Bootviren spricht, daß heute fast jeder PC mit Festplatte ausgerüstet ist und somit fast nie von Diskette urgeladen werden muß. Andererseits passiert gerade dies recht häufig, wenn beim Ausschalten versehentlich eine Diskette im Laufwerk vergessen wird. Resultat ist, daß Bootviren sich immer noch erfolgreich verbreiten und deshalb weiter entwickelt werden. Auf Fehler des Anwenders kann der Virenprogrammierer stets bauen; ein wichtiger Aspekt bei der Planung von Schutzmaßnahmen.

Bootviren sind prinzipiell kein PC-spezifisches Problem, da jeder Rechner zunächst sein Betriebssystem von einem externen Speichermedium lesen muß. Nur ist es z.B. bei Mainframes viel schwieriger, an die notwendigen Informationen heranzukommen, die Bootsoftware sinnvoll zu verändern (hohe Anforderungen an den Programmierer) und die relevanten Datenträger zu manipulieren (Zugangssicherungen etc.). Beim PC ist die Bootsoftware sehr simpel gehalten, notwendige Informationen sind in jedem Programmierhandbuch verfügbar und urladefähige Datenträger (Disketten) befinden sich in großer Anzahl im Umlauf.

Spawning Viruses. Unter MS-DOS existiert ein weiterer, auf Eigenheiten des Betriebssystems spezialisierter Virustyp, die Spawning Viruses (engl. to spawn[4] = laichen). Das Virus nutzt die Suchstrategie von `command.com` aus, wenn ein Programm gestartet werden soll: Bei gleichen Namen ist die Reihenfolge der Auswahl `com` – `exe` – `bat`. Spawning Viruses "infizieren" `exe`-Programme, indem sie ein gleichnamiges, verstecktes `com`-Programm anlegen, welches das eigentliche Virus darstellt. Dieses wird bei einem vermeintlichen Start des Originalprogramms zuerst ausgeführt und übernimmt die Kontrolle. Streng genommen handelt es sich bei dieser Softwareanomalie um keinen Virus, denn das ursprüngliche Programm wird nicht verändert. Dafür ist die Definition eines Trojaners erfüllt, denn der Start des Programms bewirkt nicht die Ausführung des erwarteten Programms. Die Bezeichnung "Spawning Virus" hat sich bereits eingebürgert, deshalb soll sie auch hier weiter verwendet werden.

[4]Die UNIX-Funktion `spawn` teilt den Programmablauf in zwei parallele Tasks.

1.3.5 Tarnung

Durch Tarnung versuchen Viren ihre Anwesenheit und ihre Aktivitäten vor dem Anwender und Prüfprogrammen zu verbergen. Bei der Infektion eines Programms wird eine Reihe von Dateieigenschaften verändert: der Inhalt, das Datum und die Zeit des letzten Schreibzugriffs sowie evtl. die Länge sowie, quasi optional, Dateiattribute (Schreibschutz etc). Indirect Action-Viren verringern durch ihre speicherresidente Installierung die Größe des freien Arbeitsspeichers und müssen Interruptvektoren umsetzen. Die in den beiden folgenden Abschnitten angesprochenen Tarnmaßnahmen erfordern z.T. Zugriff und Manipulation auf Betriebssystemebene, was auf Homecomputern und PCs aber keine unüberwindliche Hürde darstellt.

Passive Tarnung. Wenn mit Systemaufrufen Dateien manipuliert werden, aktualisiert das Betriebssystem im Verzeichnis die aktuelle Länge der Datei sowie Datum und Zeit (*Time Stamp*) des letzten Schreibzugriffs. Viren wie z.B. das *"Hallöchen"*-Virus (MS-DOS) erhalten jedoch die Zeitmarke, obwohl sie zur Infektion schreibend auf Programmdateien zugreifen müssen. Andere Viren können das Betriebssystem, welches das Dateiverzeichnis im normalen Betrieb auf dem aktuellen Stand hält, komplett umgehen und so auch die Länge scheinbar beibehalten. Eine weitere Methode ist die Komprimierung des Wirtsprogramms auf eine Länge, die zusammen mit der des Viruscodes die ursprüngliche Größe der Datei ergibt. Diesen Aktionen ist gemeinsam, daß sie nur zum Zeitpunkt der Infektion durchgeführt werden. Die Tarnung erfolgt einmalig (s.a. Direct Action) und muß später nicht aktiv (→ Name!) aufrechterhalten werden.

Aktive Tarnung. Die neueste Generation von Computerviren benutzt sog. *Stealth*-Techniken, um ihre Anwesenheit vor dem Anwender zu verbergen. Die Tarnung wird durch Übernahme und Manipulation von Betriebssystemfunktionen ständig aufrechterhalten (s. *Indirect Action*). Dazu als Beispiel das "4096"-Virus. Dieses Virus installiert sich speicherresident und klinkt sich in verschiedene Betriebssystemfunktionen ein. Bei Eröffnung einer Datei (`open`-Kommando), z.B. durch ein Anti-Virus-Programm, prüft das Virus, ob es sich um ein verseuchtes Programm handelt. Ist dies der Fall, wird das Programm desinfiziert und das Detektionsprogramm findet ein scheinbar unverändertes Programm vor, aber nur, solange das Virus aktiv im Speicher ist — ein scheinbar paradoxes, aber interessantes Phänomen. Wenn der Zugriff nach der Überprüfung abgeschlossen wird (`close`-Kommando), reinfiziert das Virus das Programm. Andere Viren nisten sich im Bootblock von Disketten und Festplatten ein und speichern den ursprünglichen Inhalt an anderer Stelle ab. Bei einer Schreib- oder Leseanforderung für den Bootblock wird der Zugriff, vom aufrufenden Programm unbemerkt, auf das Original umgelenkt.

Festzuhalten ist: Unabhängig davon, ob sich Länge oder Zeitmarke einer Datei verändert haben, ist der Inhalt der Datei nach einer Virus-Infektion **auf jeden Fall** vom Original verschieden.

1.4 Die Bedrohung durch Softwareanomalien

Allen drei Typen ist gemeinsam, daß ihre bloße Existenz bereits Speicherplatz und Rechenzeit konsumiert. Weitergehende Fähigkeiten hängen davon ab, welche Art von "Nutzlast" ihnen vom Programmierer mitgegeben worden ist. Die Palette reicht von "nur Verbreiten" über "Paßwörter stehlen" bis "Hardware zerstören". Fehler in der Programmierung sorgen überdies dafür, daß auch Viren, die von der Intention her harmlos sind, Schaden anrichten können (z.B. "nVir" auf dem Apple MacIntosh). Zusammenfassend ist zu sagen, daß die Anwesenheit solcher Programme in Rechnersystemen in höchstem Maße unerwünscht ist. Cohen schätzt in [21] die Gefahr folgendermaßen ein:

> "As an analogy to a computer virus, consider a biological disease that is 100% infectious, spreads whenever animals communicate, kills all infected animals instantly at a given moment, and has no detectable side effects until that moment."

Die möglichen Schäden lassen sich grob in drei Gruppen einteilen, die der nachfolgende Text behandelt.

1.4.1 Denial of Service

Der Verbrauch von Ressourcen und das Blockieren von Dienstleistungen wird mit dem Fachausdruck *Denial of Service* umschrieben. Von Softwareanomalien verbrauchte Rechenzeit und Speicherplatz steht anderen Programmen, und damit den dahinter stehenden Benutzern, nicht mehr zur Verfügung. Dazu kommen Kosten, die für die Wiederbeschaffung der Daten und Säuberung der Rechner anfallen. Dieser Posten stellt bisher den Löwenanteil der Schäden dar, die Softwareanomalien verursachen [89].

Darüber hinaus gibt es Viren, die gezielt Dienstleistungen behindern. Das "Jerusalem"- und das "Hallöchen"-Virus z.B. stehlen Rechenzeit, indem sie den Rechner durch Einlegen von Warteschleifen immer mehr verlangsamen. Andere Softwareanomalien verfälschen oder blockieren die Ausgabe über Drucker, lassen Buchstaben vom Bildschirm rieseln ("Cascade"-Virus) oder Verschlüsseln den Inhalt der Harddisk (AIDS-Trojaner).

1.4.2 Software-Schäden

Viren können Daten auf vielfältige Art vernichten oder, was fast noch schlimmer ist, verfälschen. Wird eine Datei von einer Softwareanomalie vollständig vernichtet, muß diese entweder neu erstellt oder vom hoffentlich vorhandenen und möglichst aktuellen Backup (Sicherheitskopie) nachgeladen werden. Verfälschungen dagegen können so subtil erfolgen, daß die schleichende Veränderung der Daten erst nach Tagen, Wochen oder gar Monaten bemerkt wird. Der unmittelbare Schaden liegt in der Verwendung der falschen Daten, die z.B. durch die Verschiebung von Kommastellen in einer Kalkulation verheerende wirtschaftliche Folgen für ein Unternehmen nach sich ziehen kann.

Als zusätzliche Erschwernis befinden sich durch die lange Zeit, über die hinweg keine Veränderungen bemerkt wurden, die falschen Daten wahrscheinlich auch auf den Sicherheitskopien.

Nicht zu vergessen sind überdies die immateriellen Schäden, die durch den Verlust von Vertrauen in das System und Datenbestände entstehen. Wer möchte sein Konto bei einer Bank behalten, von der bekannt wird, daß sie Opfer eines Virenangriffs wurde? Was sind aufwendig beschaffte, geheime Daten noch Wert, wenn sie möglicherweise manipuliert und weitergegeben wurden? Bloße Mund-zu-Mund-Propaganda kann einem Unternehmen mehr schaden, als es ein Virus je vermocht hätte. Nicht zuletzt aus diesem Grund dürfte die Dunkelziffer für Attacken durch Softwareanomalien recht hoch liegen. Einige Firmen wie z.B. die BASF hingegen werben geradezu damit, daß sie die Zeichen der Zeit erkannt haben und treten mir ihren Schutzaktivitäten an die Öffentlichkeit heran [**?**]. Dies ist eine Methode, die wahrscheinlich mehr Vertrauen schafft, als die — vermutlich inkorrekte — Behauptung, "bei uns kann so etwas gar nicht passieren".

Wenn sich die Manipulation der Daten bei der Untersuchung des verursachenden Programms als irreversibel erweist, bleibt nur noch das Löschen der fehlerhaften Datensicherung und die Neuerstellung, falls diese überhaupt möglich ist. Reversible Beschädigungen lassen sich nach Analyse der Schadensfunktion wieder rückgängig machen. Dennoch sollte, falls möglich, das Original von der Sicherheitskopie zurückgeladen werden, um mögliche bei der Reparatur gemachte Fehler und das Wiedereinspielen übersehener verseuchter Programme zu vermeiden.

1.4.3 Hardware-Schäden

Viren können nicht nur Daten vernichten, sondern auch die Hardware eines Rechners beschädigen. Diese Möglichkeit wird oft bezweifelt; die folgenden Beispiele beweisen aber das Gegenteil. Auf dem Commodore C-64 beispielsweise gab es ein Programm, das mit dem Diskettenlaufwerk eine Melodie spielte, indem es den Lesekopf entsprechend schnell gegen den Endanschlag vibrieren ließ. Dies war der empfindlichen Laufwerksmechanik sicher abträglich. Allerdings machte der Aufbau des Laufwerks noch eine Steigerung möglich. Da nur ein Endanschlag vorgesehen war, konnte der Kopf durch entsprechende Befehle an den Laufwerks-Controller bis auf die Diskettenhülle verschoben werden. Es geht aber auch subtiler und weniger auffällig: Ein Virus könnte jedem Zugriff auf Platte oder Diskette zusätzliche Bewegungen des Schreib-/Lesekopfes hinzufügen und dadurch den Verschleiß drastisch erhöhen (Nathanson, Larry in [38] 4.024).

Ein anderes Beispiel ist das unterdimensionierte Netzteil, mit dem die ersten der 1985 erschienenen tragbaren Computer der Firma Compaq ausgestattet waren. Dieses versorgte neben dem eigentlichen Rechner auch den Monitor mit Strom. Bei falscher Programmierung des Videocontrollers konnte es dazu kommen, daß die Bildwiederholfrequenz und damit der vom Monitor verbrauchte Strom groß genug wurde, um eine Sicherung im Netzteil durchbrennen zu lassen. Da diese nur schwer erreichbar und auf der Platine aufgelötet war, wurde deren Ersatz zu einer aufwendigen Aktion (Bosen, Bob in [38] 4.034). Bei den ersten "Monochrome Adapter Cards" von IBM ließ sich

die horizontale Ablenkfrequenz für den Schreibstrahl auf null setzen. Die intensive Bestrahlung nur einer einzigen Zeile beschädigte auf Dauer die Leuchtschicht des Monitors (Murray, William Hugh in [38] 3.157).

1.5 Historie und Dynamik der Entwicklung

1.5.1 Die Experimente und Theorien von Fred Cohen

Der erste, der sich mit Computerviren wissenschaftlich auseinandersetzte, war Fred Cohen. Sein Aufsatz "Computer Viruses - Theory and Experiments" [21] erschien 1984, 1986 folgte eine Doktorarbeit über das gleiche Thema. Cohen definiert den Begriff "Computervirus" wie folgt:

> "A computer 'virus' ... [is] a program that can 'infect' other programs by modifying them to include a possibly evolved copy of itself."

In dieser vorausschauenden Definition sind auch mutierende Viren, die sich nicht nur verbreiten, sondern auch selbst modifizieren, mit eingeschlossen, obwohl dieser Typ erst Ende 1990 zum ersten Mal aufgetreten ist. Das 2. Kapitel beschäftigt sich noch ausgiebig mit den von Cohen entwickelten Theorien zur Abwehr von Computerviren. Doch nun zu den Experimenten, die Cohen Mitte der 80er Jahre durchgeführt hat [22].

Datum	Ereignis
03.11.83	Entwicklung eines Virus unter UNIX (VAX 11/750)
10.11.83	Experiment mit über Bulletin Board eingeschleustem verseuchtem Programm
	Tests unter Tops-20, DEC VMS, IBM VM/370
07.84	Test auf Bell-LaPadula-System (Univac 1108)
08.84	Test unter UNIX (VAX)

Tabelle 1.1: Cohen's Experimente auf Großrechnern

Für die Programmierung des ersten, sehr einfachen Virus benötigte ein Experte nur acht Stunden. Dennoch gelang es in fünf einzelnen Versuchen, in durchschnittlich einer halben, höchstens einer Stunde, Supervisor-Rechte zu erhalten. Auch für die Viren auf den anderen Systemen wurde selbst im Extremfall eine Entwicklungszeit von nur 30 Stunden benötigt. Es gelang jeweils immer, in relativ kurzer Zeit (Sekunden bis wenige Stunden) die Grenzen zwischen Benutzern und Sicherheits-Ebenen zu überwinden. Dabei wurde weder das Sicherheitssystem umgangen noch irgendwelche Mängel ausgenutzt.

Ein sehr interessantes und oft kontrovers diskutiertes Thema ist, ob ein Virus Sicherheitseinrichtungen des Betriebssystems umgehen muß oder nicht. Man liest oft die Behauptung, daß die Entwicklung von Computerviren auf Minis und Mainframes ungleich schwieriger als auf z.B. MS-DOS-Rechnern sei; zum einen, weil die Sicherheitseinrichtungen der Ausbreitung im Wege stehen und zum anderen, weil systemnahe

Programmierung auf Großrechner-Betriebssystemen erhebliche Programmierfertigkeiten (engl. skill) erfordert.

Beschränkt man sich, wie Cohen in seinen Experimenten, auf rel. simple Viren und auf völlig legale Systemoperationen, sind trotzdem bis auf Supervisorebene durchschlagende Erfolge möglich. Unter UNIX zum Beispiel erbt ein Programm die Rechte des Benutzers, der es gestartet hat. Startet Benutzer A ein verseuchtes Programm, kann das Virus nur auf die Dateien schreibend zugreifen, für die A auch Schreiberlaubnis hat. Nun kann A seine Programme dem Benutzer B, jedem anderen seiner Benutzergruppe oder allen Rechnerbenutzern zugänglich machen. Startet B das infizierte Programm, kann das Virus alle anderen Programme verseuchen, auf die B Schreibzugriff hat. Die Benutzung eines Programms durch mehrere Anwender nennt man "sharing" (von engl. to share = teilen, teilhaben). Cohen stellte nun ein den anderen Benutzern unbekanntes, aber als interessant angepriesenes Programm namens `vd` zur Verfügung. Innerhalb von Minuten probierte auch der Systemadministrator (Benutzerkennung "root") das Programm aus. Da UNIX bei der Systemverwaltung keine Gewaltenteilung kennt, hatte das Virus damit alle Systemrechte inne.

Fazit: Computerviren, die aus Gründen der Tarnung professionelle, mit Sicherheitsmechanismen ausgestattete Betriebssysteme umgehen, sind in der Tat schwierig zu schreiben. Einfache und trotzdem gefährliche Viren hingegen sind schnell und ohne große Programmiererfahrung zu erstellen und bei der Verbreitung sowie bei der unbefugten Erlangung von Rechten als erfolgreich einzustufen.

1.5.2 Die Verbreitung von Viren bis heute

Cohen veröffentlichte seine Arbeiten am 16.10.84 an der University of Southern California. Schon vorher, 1983, sprach Len Adleman im Zusammenhang mit Cohen's Experimenten von Computerviren. Bereits am 19.11.84 erschien im Nachrichtenmagazin "Der Spiegel" ein Artikel zu diesem Thema, der sich allerdings nur sehr oberflächlich mit Computerviren befaßte. In der Ausgabe 3/'85 der "Bayerischen Hacker Post" (BHP) fand sich eine ausführliche Zusammenfassung von Cohens Erkenntnissen; die — übrigens sehr empfehlenswerte — Sicherheitszeitschrift "KES" folgte trotz einiger Skrupel, denn es wurde die Frage diskutiert, ob man derartige Kenntnisse überhaupt öffentlich verbreiten dürfte oder sollte. In der Computerwoche vom 13.09.85 bezeichnet ein Artikel den BHP-Aufsatz als Straftat und rückt die Autoren in die Nähe von Terroristen [19].

Danach lockerte sich die Sachlage etwas auf. Die Zeitschrift "Apples" druckte am 11.12.85 einen Virus als Listing ab; auch die "Computer Persönlich" in ihrer Ausgabe Nr. 24 vom 12.11.86 steuerte neben einem interessanten Artikel über die Anfänge der Computerviren ein Exemplar für den Apple II bei. Beiträge über Computerviren erschienen im "PM-Computerheft", in der "Chip", der "Süddeutschen Zeitung" und im Zeitgeistmagazin "Tempo". Allerorten wurden Quelltexte von Viren veröffentlicht (z.B. das "Cookie"-Virus); eine ärgerliche Unart, die man heute noch in manchen Publikationen antreffen kann [27].

1986 war es dann soweit, daß Viren begannen, sich unangenehm bemerkbar zu machen. Die FU Berlin mußte sich gegen Viren rüsten, die von Studenten eingeschleppt wurden [20]. Panikmeldungen geisterten durch die Presse, Aktivierungsdaten von Computerviren wurden wie der jüngste Tag der Computer gehandelt [53].

Abbildung 1.3: Von `ViruScan` erkannte Viren

Bis Mitte 1989 war es trotz einiger Vorfälle auf dem Virensektor vergleichsweise ruhig, etwa 77 Typen waren bekannt. Ende 1989 nahm die Entwicklung eine Wendung, die man getrost als dramatisch bezeichnen kann und die bis heute (Anfang 1992) anhält. Die Anzahl der Virentypen erhöhte sich von Monat zu Monat, die Autoren von Abwehrprogrammen hielten nur mit Mühe schritt (s.Abb. 1.3). Woran liegt das? Viren sind im Prinzip einfach zu programmieren und erfordern keine Spezialinformationen, die sich nicht aus ganz normalen Betriebssystemunterlagen des Herstellers entnehmen ließen. Informationen zur Funktion und Programmierung wurde auf breiter Ebene für jedermann verfügbar. Bei der Erstellung eines Virus geht es tatsächlich weniger um das technische "Kann ich das?" als um das ethische und rechtliche "Will/sollte/darf ich das?". Nicht zuletzt durch die Aufmerksamkeit, die den Viren durch die Medien entgegengebracht wird, besteht für offensichtlich immer mehr Menschen ein Anreiz, sich an die Virenprogrammierung zu wagen.

Der Großteil der Viren sind einfallslose Modifikationen rel. weniger Grundlinien (engl. strains). Insbesondere Viren, deren Quelltext in Zeitschriften oder über BBS (Bulletin Board Systems = öffentliche elektronische Tauschbörse für Programme) veröffentlicht wurde, sind Gegenstand dieser Art von "Programmierung", die den Schöpfer von eigener geistiger Tätigkeit weitgehend entbindet. Zur Not könnte der Kopierer auch ein Virus selbst entschlüsseln und verändern ("Reverse Engineering").

Einem automatischen Mutationsgenerator, der Unterroutinen des Virus zufällig mischt und verschlüsselt, steht leider nichts außer dem vielleicht in Relation zum Geltungsbedürfnis unterentwickelten eigenen Gewissen und Verantwortungsbewußtsein entgegen. Selbst der kleinen Freude eines selbstentwickelten Virusgenerators, der aus einer Anzahl von vorgefertigten Modulen auf Anweisung ein einsatzfähiges Virus konstruiert, kam eine Firma mit dem "Virus Construction Set" für den ATARI ST zuvor. Derlei Werkzeuge versetzen auch des Programmierens gänzlich Unkundige in die Lage, Viren zu produzieren.

Die Frage, wie schnell sich ein bestimmtes Virus verbreitet, ist Gegenstand verschiedener Untersuchungen. Da jeder Wurm und jedes Virus eine bestimmte Anzahl von Nachkommen erzeugt und jeder dieser Nachkommen sich selbst wieder vermehrt, müßte die Verbreitung theoretisch exponentiell erfolgen. Schätzungen über die tatsächliche Ausbreitungsgeschwindigkeit variieren stark untereinander. Da viele unbekannte Faktoren wie tatsächliche Vermehrungsrate, Austausch von Vektoren, Elimination und andere in die Rechnung eingehen, ist das Ergebnis entsprechend ungenau. Tatsache ist, daß heute für ein neu entdecktes Virus binnen weniger Tage ein Abwehrprogramm entwickelt wird. Verbreitungsraten wie aus den Gründerzeiten der Computerviren (z.B. "Brain", ein Boot-Virus) gehören der Vergangenheit an. Dennoch sollte nicht unterschätzt werden, daß viele, wahrscheinlich die allermeisten Computer nur unzureichend geschützt sind und längst nicht alle möglichen Maßnahmen auch ergriffen werden. Dazu kommt neben technischen Aspekten oft auch die mangelnde Kenntnis der Anwender über das vorhandene Sicherheitsrisiko. Dadurch wird auch den einfachsten Computerviren das Vordringen ermöglicht.

Die durch die angesprochenen Möglichkeiten und Verfahren erzeugte Virenflut erschwert verantwortungsvollen Programmierern die Entwicklung von Programmen, die Viren erkennen und beseitigen können. Es zeichnet sich eine Entwicklung ab, die die Bekämpfung bestimmter, d.h. identifizierbarer Viren unmöglich macht. Konzepte zur Abwehr von Softwareanomalien allgemein und Computerviren im speziellen sind Gegenstand des nächsten Kapitels. Vorher soll zunächst noch die rechtliche Seite der Problematik und Maßnahmen von öffentlicher Hand untersucht werden.

1.6 Maßnahmen öffentlicher Stellen

Die mutwillige Verseuchung von Rechnern kann u.U. auch rechtliche Konsequenzen nach sich ziehen. Allerdings steht bei Softwareanomalien einer strafrechtlichen Verfolgung und Beweisführung entgegen, daß fast grundsätzlich der Urheber nicht ermittelt werden kann. Dem Programmierer genügt es, an einer für andere Benutzer erreichbaren Stelle das Manipulationsprogramm zu installieren; die weitere Verbreitung erfolgt unbewußt durch ahnungslose Benutzer, die das Programm auf Wegen transportieren, die der Kontrolle durch Sicherheitssoftware entzogen sind. Vor allen Dingen die auf dem Homecomputer- und PC-Sektor weit verbreitete Praxis der illegalen Weitergabe von Raubkopien macht eine nachträgliche Feststellung des Infektionsweges unmöglich.

Selbst wenn der Verursacher feststellbar ist, entstehen aus Mangel an eindeutigen

Gesetzen Probleme, wie sie aus der prä-elektrischen Zeit bekannt sind. Der Diebstahl von elektrischer Energie konnte nicht betraft werden, weil Strom keine "fremde, bewegliche Sache" im Sinne des Gesetzestextes war. Gemäß dem Prinzip "nulla poena sine lege", keine Strafe ohne Gesetz, lag kein Straftatbestand vor, nach dem Recht hätte gesprochen werden können. Während in den USA in neuerer Zeit Gesetze erlassen wurden, die sich ganz konkret mit Softwareanomalien befassen, ist die Rechtsgebung in Deutschland deutlich hinter den Gegebenheiten zurück und wird es vermutlich auch noch für eine ganze Weile bleiben. Wie bei allen Hi-Tech-Verbrechen ergeben sich auch bei Softwareanomalien zusätzliche Schwierigkeiten durch die Tatsache, daß Richter und Jury in der Regel Laien auf dem Gebiet sind, das Gegenstand des Verfahrens ist. Im Fall des "Morris-Wurms" kam in den USA in den Medien und auf der Diskussionsliste `VIRUS-L` eine Diskussion darüber auf, ob Richter und Jury Laien, sachverständig oder gemischt sein sollten; eine Frage, die nicht mit einem Satz zu beantworten war und von offizieller Seite bis heute nicht geregelt ist.

1.6.1 Die Rechtslage in Deutschland

1986 wurde das "zweite Gesetz zur Bekämpfung der Wirtschaftskriminalität" in Kraft gesetzt, in dem sich fünf Paragraphen mit der Computerkriminalität allgemein befassen, aber keiner speziell mit Softwareanomalien. Am ehesten auf Viren zutreffend sind die Paragraphen §303a "Datenveränderung" und §303b "Computersabotage" StGB.

Unter "Datenveränderung" wird verstanden, Daten zu löschen, zu unterdrücken, unbrauchbar zu machen oder zu verändern. Der "Computersabotage" macht sich schuldig, wer Daten verändert oder eine DV-Anlage oder einen Datenträger zerstört, beschädigt, unbrauchbar macht, beseitigt oder verändert. Das Strafmaß bewegt sich zwischen 2 und 5 Jahren Freiheitsentzug oder Geldstrafe. Softwareanomalien verstoßen in mindestens einem Punkt gegen eines der oben genannten, für strafrechtlich relevant befundenen Kriterien. Trotzdem lassen sich mit diesen Gesetzen Computerviren — wenn überhaupt — nur schwer fassen. Problematisch ist insbesondere, daß nur der Begriff "Daten", nicht aber der Begriff "Programm" erwähnt wird. Auch andere Gesetzestexte sind nicht sehr hilfreich. Der Paragraph §202a StGB stellt das "Ausspähen von Daten" nur dann unter Strafe, wenn die Information "gegen unberechtigten Zugriff besonders gesichert" ist, wovon bei PCs so gut wie nie die Rede sein kann. Nach der deutschen Rechtslage können Virenprogrammierer nur bei weitestmöglicher Auslegung der Gesetze bestraft werden [26].

1.6.2 Die Rechtslage in den USA

In den USA wurde 1988 ein Gesetzesentwurf eingebracht, der sich speziell mit der Problematik der Computerviren befaßt (HR-5061: "Computer Virus Eradication Act of 1988" [26]; "Gesetz zur Ausrottung von Computerviren"). Das Gesetz droht demjenigen Strafe bis zu 10 Jahren Freiheitsentzug an, der wissentlich Programme oder Daten so manipuliert, daß diese ihren Benutzern Schaden zufügen. Es genügt dabei bereits, solche Programme unerkannt in Umlauf zu bringen oder dies zu versuchen.

Ein Jugendlicher wurde bereits dafür bestraft, daß er absichtlich einen Virus in ein BBS eingeschleust hat [92]. Ebenfalls verurteilt (zu $10000 Strafe, 400 Stunden gemeinnütziger Arbeit und einem Jahr Ausschluß von der Universität) wurde Robert T. Morris, dessen INTERNET-Worm 1990 Hunderte von zivil und militärisch genutzten Rechnern blockierte und neben Aufsehen in den Medien über 60 Millionen Dollar Schaden (geschätzt) verursachte [68].

1.6.3 Sicherheit von IT-Systemen

Das BSI. Auf der anderen, prophylaktischen Seite werden von öffentlichen Stellen Anstrengungen unternommen, Methoden und Verfahren zu entwickeln, um Rechnersysteme allgemein sicherer zu machen. Die ZSI (Zentralstelle für Sicherheit in der Informationstechnik; ehem. ZCh Zentralstelle für Chiffriertechnik) veröffentlichte in Zusammenarbeit mit Behörden, Verbänden und der Industrie am 11.1.89 Kriterien zur Bewertung von IT-Systemen (Systemen der Informationstechnik), die im Februar 1990 um ein Handbuch ergänzt wurden [82]. Die Grundkriterien sind Vertraulichkeit, Verfügbarkeit und Integrität. Anhand dieser werden die zu bewertenden Systeme aus den Bereichen Großrechner, mittlere Datentechnik, APC (Arbeitsplatzcomputer[5])/Workstations und Netzwerke in F- (Funktionalitäts-) Klassen und Q- (Qualitäts-) Klassen eingeteilt. Es werden dabei folgende Produkttypen unterschieden:

- Zugriffsschutz (für bel. Ressourcen)
- Kopierschutz
- Bootschutz
- Zugriffsschutz (für den APC als Gerät, z.B. Sicherheitsschloß etc.)
- Verschlüsselung
- Virusschutz und -erkennung
- Logging (Protokollierung)
- sonstige Sicherheitsfunktionen (Datensicherung, Authentifikation, sicheres Löschen von Dateien etc.)

Die ZSI wurde am 1.1.1991 in das BSI (Bundesamt für Sicherheit in der Informationstechnik) umgewandelt, das direkt dem Ministerium des Inneren unterstellt ist [84].

Es laufen zur Zeit (Mitte 1991) Bemühungen seitens Deutschland, England, Frankreich und den Niederlanden, einen zumindest europaweiten Konsens zu finden. Auf amerikanischer Seite wurden nach Veröffentlichung der "Grünbücher" Proteste laut, die sich gegen Wirtschaftsprotektionismus per Sicherheitskriterien wandten. Gleiche Vorwürfe ließen sich natürlich auch auf europäischer Seite gegen das "Orange Book" erheben. Erstrebenswert erscheint daher die Ausarbeitung von Konvertierungsregeln, die die Gleichwertigkeit der verschiedenen Sicherheitsstufen festlegen.

[5] APC ist eine allgemeinere Bezeichnung für am Arbeitsplatz eingesetzte Computer als PC.

Das Orange Book. Das DoD (Department of Defense; Verteidigungsministerium der USA) veröffentlichte bereits 1983 seine "Trusted Computer System Evaluation Criteria", Kriterien zur Bewertung vertrauenswürdiger Computersysteme, im sog. "Orange Book" [**?**]. Anhand dieser Bewertungsmaßregeln wird ein Computersystem in bestimmte Klassen eingeordnet, die stark an militärische, hierarchische Sicherheitsstufen angelehnt sind (Tab. 1.2). Diese Einstufung gilt nur für eine bestimmte Konfiguration der Software auf einer bestimmten Hardware und ist Voraussetzung für den Einsatz eines Systems im militärischen Bereich. Da weder das Konzept noch die Kriterien selbst zur Bewertung von komplexen oder vernetzten System ausreichen und das Bewertungssystem im Ganzen zu unflexibel ist, wird seit 1985 an einem "Red Book" gearbeitet. Auch zivile Behörden wie das NIST (National Institute of Standards and Technologie) bemühen sich seit 1987 um flexiblere und auf zivile Anforderungen zurechtgeschnittene Kriterien.

Level	Bedeutung
D	Unsicheres System
Levels mit willkürlicher Kontrolle (DAC = **D**iscretionary **A**ccess **C**ontrol); der Anwender kann den Zugriff auf seine Daten festlegen	
C1	Anwender muß sich einloggen; Benutzergruppen erlaubt
C2	Anwender muß sich einloggen; Paßwort und Logdatei erforderlich
Levels mit obligatorischer Kontrolle (MAC = **M**andatory **A**ccess **C**ontrol); der Zugriff wird durch DoD-Richtlinien festgelegt	
B1	Freigabeebenen nach DoD
B2	System ist testbar; kein Fluß von Information zu niedrigeren Freigabeebenen möglich
B3	System wird durch mathematisches Modell beschrieben
A1	System wird durch beweisbares mathematisches Modell beschrieben (höchste Sicherheit)

Tabelle 1.2: Sicherheitsstufen des Orange Book

Mittlerweile wurden eine ganze Reihe von Betriebssystemen und Sicherheitsprodukten durch das NIST bewertet oder befinden sich im Evaluationsprozeß, für den etwa zwei Jahre zu veranschlagen sind. Die Betriebssysteme VAX VMS 4.3 und die auf Großrechnern verbreiteten Sicherheitsprodukte ACF2 und TOP SECRET von Computer Associates sowie MVS-ESA/RACF von IBM besitzen die Einstufung C2; Computer Associates und IBM streben das B1-Level für ihre Produkte an. AT&T wartet mit einem B1-UNIX mit der Bezeichnung "System V/MLS" (für Multi Level Security) auf, das auch für die Stufen C1 und C2 konfiguriert werden kann.

Damit bei neuen Versionen nicht eine komplette Neubewertung durchgeführt werden muß, wurde das RAMP- (**ra**ting **m**aintenance **p**hase-) Programm ins Leben gerufen, mit dessen Hilfe eine einmal erreichte Sicherheitsstufe beibehalten werden soll. Z.B. arbeitet DEC an einer C2-Level MVS-Version, die Mitte 1990 in das RAMP-Programm aufgenommen wurde.

1.6.4 Organisationen zur Virusbekämpfung

1991 wurden auf Europaebene die Institutionen CARO (**C**omputer **A**nti-Virus **R**esearchers **O**rganisation) und EICAR (**E**uropean **I**nstitute for **C**omputer **A**nti-Virus **R**esearch) ins Leben gerufen. CARO ging aus einem vom Hamburger Percomp-Verlag (Anbieter des "Virus Telex"-Services) veranstalteten internationalen Virenforum hervor. Das Microbit Virus Center der Universität Karlsruhe soll demnach als Sammelstelle für neu entdeckte Viren und Virusinformationen dienen und Anti-Virusforschern zur Verfügung stehen. Unter Beteiligung von Industrie, Staat und Wissenschaft wurde EICAR zur Lösung von praktischen Problemen bei der Virenbekämpfung gegründet.

Von dem durch CARO bereitgestellten Informationspool wurden Personen ausgeschlossen, die Viruscode publiziert und damit zur weiteren Ausbreitung von Softwareanomalien beigetragen haben. Dazu zählen Antivirus-Forscher wie John McAfee (Programm u.a. `ViruScan`) und Prof. Eberhard Schöneburg (Buch "Computerviren" [35]) sowie der Buchautor Ralf Burger ("Das große Computer-Viren Buch" [27]). Prof. Schöneburg hat sich bereits energisch gegen den Ausschluß und ein angebliches Informationsmonopol von BSI und "ihm genehmen Sicherheitsexperte[n]" gewehrt [**?**].

Der KES-Artikel schließt mit der nur zu wahren Bemerkung, daß polemische Kleinkriege zwischen Forschern zwar nichts Neues sind, aber zur erfolgreichen Virenbekämpfung nichts beitragen. Angesichts der zunehmenden Verknüpfung von Wirtschaft und Universitäten auch in Deutschland und den damit ins Spiel kommenden Geldern ist mit weiteren, wirtschaftlich motivierten Streiten und Eifersüchteleien zu rechnen.

Kapitel 2

Theorie der Abwehr

> Short of using the U.S. military's M-16 security solution (a U.S. Marine with an M-16 rifle outside the computer room that does not have any dial-up lines), it is next to impossible to secure any system fully.
>
> Bernard P. Zajac Jr. [24]

Das 1. Kapitel beschäftigte sich ausführlich auf theoretischer Basis mit den Funktionsprinzipien der Softwareanomalien. Aus dieser Analyse der "Software-Krankheit" werden hier Abwehrstrategien und -techniken entwickelt, die in die Schutzprogramme des 4. Kapitels eingehen.

2.1 Analyse: Ursachen der Verseuchung

Wie in 1.4 angesprochen, gefährden Computerviren Programme, Daten und Hardware. Neben nicht meßbaren Schäden wie Vertrauensverlust entstehen konkrete Kosten durch den Verbrauch von Ressourcen wie Speicherplatz und Rechenzeit. All dies läßt den Wunsch nach Schutz vor und Bekämpfung von Softwareanomalien entstehen. Prinzipiell besteht die Möglichkeit, diese als Schicksalsschläge wie Diebstahl, Sabotage, Blitz oder Hochwasser hinzunehmen und konventionelle Maßnahmen zur Wiederherstellung der Daten zu ergreifen.

Dagegen gibt es allerdings einige gewichtige Einwände. Das Rückladen der zerstörten Daten von Sicherheitskopien (Backups) erfordert Personal und Zeit, die für planmäßige Anwendungen nicht zu Verfügung steht. Blieb das Virus lange unbemerkt, könnte auch das Backup infiziert sein. Schließlich wurden vielleicht nicht alle Daten gesichert; die Neubeschaffung wäre unvermeidlich. Zusätzlich stellen Softwareanomalien auch ein Sicherheitsrisiko dar, da sie zum unbefugten Erlangen von Zugriff auf vertrauliche Daten benutzt werden können. So war es z.B. dem "INTERNET-Worm" möglich, sich durch Ausprobieren von Paßwörtern von System zu System verbreiten und eine

Liste erfolgreicher Einbrüche an den Verursacher zurückzuschicken. Ebenso sind Softwareanomalien denkbar, die nach einem erfolgreichen Einbruch Hintertüren installieren und so ihrem Schöpfer beliebigen Zugriff auf das attackierte System ermöglichen.

Deshalb erscheint eine Abwehr von Softwareanomalien erstrebenswert und rentabel. Dieses Kapitel beschäftigt sich ausführlich mit der Herkunft von Softwareanomalien und ihrer Verbreitung, um Strategien zur Abwehr zu entwickeln. Diese werden zusammen mit den im Kapitel "Systemprogrammierung unter MS-DOS" vermittelten Kenntnissen in konkrete Programme umgesetzt. Die Abwehr von Softwareanomalien hat prinzipiell zwei Seiten: Eine menschliche, welche die Motive und das Verhalten der Programmierer und Verbreiter betrifft, und eine technische, an der sich Gegenmaßnahmen gegen die Virusprogramme selbst orientieren.

Auch ohne Sicherheitssoftware zu erstellen und zu installieren, läßt sich etwas gegen die Bedrohung durch Computerviren tun. Viren entstehen nicht von allein und verbreiten sich auch nicht ohne menschliches Zutun. Deshalb ist es interessant zu untersuchen, aus welchen Gründen Menschen Viren programmieren und Rechner unabsichtlich oder absichtlich verseuchen. Die Analyse der Funktion und Verbreitung von Softwareanomalien beschäftigt sich eher mit Aspekten der technischen Seite. Erkenntnisse aus beiden Untersuchungen lassen sich für Erziehung, Ausbildung sowie für organisatorische und programmtechnische Abwehrmaßnahmen nutzen.

2.1.1 Motive der Programmierer

Computerviren sind weit verbreitet und die Anzahl der verschiedenen Typen wird ständig größer. Alarmmeldungen über neu entdeckte Exemplare kommen fast täglich aus allen Teilen der Welt; aus reichen und armen Ländern, kapitalistischen und — seit 1991 vor allen Dingen — sozialistischen Staaten. Hinter der momentan erlebten Virenflut stehen hunderte von Programmierern, denn nicht jedes Virus findet seinen Weg zu den Entwicklern von Antivirussoftware. Wer sind diese Menschen, und aus welchem Grund schreiben sie absolut nutzlose Programme? Spaß am Experimentieren, Selbstbestätigung und Rache werden bei der Befragung von Computerstraftätern häufig als Gründe ermittelt.

Vorschriften. Wie kann mit Hilfe von Gesetzen, Benutzerordnungen für Rechenzentren und betrieblichen Vorschriften erreicht werden, daß niemand Softwareanomalien erstellt? Durch die weite Verbreitung von IBM-kompatiblen APCs auch im privaten Bereich entzieht sich die Programmierung von Softwareanomalien der Kontrolle durch Personen und Kontrollprogramme. Ähnliches gilt für das Fälschen von Banknoten im Hobbykeller. Der entscheidende Unterschied zwischen den genannten Straftaten besteht im Grad der Möglichkeit, den Täter zu ermitteln. Falschgeld muß gegen echtes Geld oder Waren umgetauscht und so in Umlauf gebracht werden. Es genügt hingegen, eine verseuchte Diskette irgendwo liegen zu lassen; irgend jemand wird sie schon aus Interesse benutzen. Dieser verseucht seinen Rechner, gibt infizierte Programme an Bekannte weiter, die wiederum... Der Verursacher ist an der ersten Verbreitung gar nicht aktiv beteiligt und wegen der vielen Zwischenstationen praktisch nicht mehr zu ermitteln.

Das Virus kann über Datennetze und Disketten mehrmals um die ganze Welt gereist sein; der Weg ist mit vernünftigem Aufwand nicht nachvollziehbar.

Fazit: Den vielen Hobbyprogrammierern kann niemand *wirksam* vorschreiben, was sie tun und lassen sollen. So, wie sich mit einem Farbkopierer Geldscheine kopieren lassen, kann eben ein Computer auch zur Virenprogrammierung eingesetzt werden. Allein dadurch, daß der Gesetzgeber solches Tun unter Strafe stellt, werden Computerviren nicht von der Bildfläche verschwinden.

Ethik in der Informatik. Angesichts der Computervirenproblematik sind auch schon Forderungen nach einer Ethik in der Informatik laut geworden [38] 3.015. In erster Linie wird dabei an eine Art Codex, an einen "Eid des Turing" gedacht, wie ihn die Berufsgruppen Ärzte ("Hippokratischer Eid") oder Ingenieure befolgen (sollten). Richtig ist wohl in jedem Fall die Forderung, daß sich die Grundeinstellung mancher Menschen zugunsten eines verantwortungsbewußteren Handelns ändern muß. Dennoch ist eine solche, von allen respektierte Ethik eher eine Wunschvorstellung, die durch neue Sicherheitskonzepte und Gesetze zu unterstützen ist. So richtete z.B. die Ikarus GmbH einen Wettbewerb für Virenprogrammierer aus [86]. Gesucht wurde nach einem Virus, der das Schutzprogramm dieser Firma unerkannt umgehen kann.

Viele Netzbetreiber und Universitäten haben Verhaltensmaßregeln erstellt, deren Befolgung einen Mißbrauch verhüten und unnötiger Belastung der Netze oder Rechnereinrichtungen vorbeugen soll (FTP[1]: Sammlung in `/ethics` auf `unma.unm.edu`). Im GAO- (**G**eneral **A**ccounting **O**ffice-) Report, der zukünftige Verwaltungsformen für das INTERNET vorschlägt, sind drei Handlungen aufgeführt, die als unethisch und nicht akzeptabel betrachtet werden [**?**]:

1. Unautorisierter Zugriff auf Ressourcen des INTERNETs.
2. Stören des bestimmungsgemäßen Betriebs des INTERNETs.
3. Verschwenden von Ressourcen, Zerstören der Integrität von gespeicherter Information, Verletzen der Vertraulichkeit von Daten.

Damit sind Zwischenfälle wie der "INTERNET-Worm", aber auch die Aktivitäten von Hackern voll abgedeckt. Die Vereinigung "Computer Professionals for Social Responsibility" (in Deutschland: "Forum Informatiker für Frieden und gesellschaftliche Verantwortung", kurz FIFF) und andere Gruppen haben sich ausdrücklich für

1. die Durchsetzung strenger ethischer Regeln,
2. Ethik-Vorlesungen für Studenten der Informatik und die
3. Wahrnehmung individueller Verantwortlichkeit

[1] Zugriff auf Rechnernetze s. Anhang B.

ausgesprochen. Das Problem wird aber weiterhin sein, daß noch so oft proklamierte ethische Verhaltensweisen nur eine Richtlinie sein können, an die sich viele nicht halten werden. Der "Chaos Computer Club" in Hamburg vertritt z.B. eine "Hackerethik", an der sich vielleicht die Mitglieder dieser Vereinigung orientieren mögen. Trotzdem passen Hacker weder in die Schublade "Kriminelle" noch "Wohltäter". Letztendlich ist jeder zu Hause am Gerät nur seinem Gewissen verantwortlich.

Was die Verletzung dieser geschriebenen und ungeschriebenen Regeln angeht, üben erfreulicherweise verschiedene Gruppen innerhalb der Netzgemeinde durchaus Druck auf verantwortliche Stellen aus. So wurde z.B. einem Netzteilnehmer aus Fernost der Zugang zum BITNET, einem der größten Rechnernetze der Welt, gesperrt, weil dieser öffentlich nach einem Virus für VAX-Systeme oder Vorschläge zu dessen Realisierung suchte [38] 3.035, 3.029.

Dergleichen verantwortliches Verhalten scheint nicht allzu verbreitet zu sein. Auf der ganzen Welt, auch in Deutschland, existieren BBS, über die nicht nur aus Nachlässigkeit verseuchte Software, sondern auch absichtlich der Quelltext von Viren verbreitet wird. Speziell im osteuropäischen und asiatischen Raum herrscht auf dem Sektor der Softwareanomalien offensichtlich Narrenfreiheit. Das mag mit daran liegen, daß entsprechende Gesetze und Behörden, um deren Übertretung zu verfolgen, fehlen. Virusprogrammierer schmücken ihre Werke ungeniert mit Namen und Adresse (Pakistan, Bulgarien) und geben Interviews, die in großen Computerzeitschriften samt Konterfei erscheinen [**?**]. Durch solches Verhalten kommt eine Verantwortungslosigkeit zum Ausdruck, deren Ursache man in Naivität, Ignoranz oder bösem Willen ausmachen kann.

Etliche Zeitschriften und Bücher, besonders aus der Goldgräberzeit der Computerviren, veröffentlichten Programmcode "fix und fertig zum Abtippen" für z.B. den Homecomputer Apple II. Während älteren Publikationen zugute gehalten werden kann, daß damals Viren noch kein erkanntes Problem waren, gilt das sicher nicht mehr für Schriften aus neuerer Zeit. Selbst Bücher, deren Aufmachung den Eindruck vermittelt, daß ihr Anliegen die Bekämpfung von Viren ist, enthalten funktionsfähigen Quelltext oder zumindest detaillierte Anleitungen in Pseudocode.

Gängige Schutzbehauptungen wie "Einsatz nur zu Testzwecken" sollen die Verantwortung des Autors auf den Leser verlagern. Einer dieser "harmlosen Testviren" ist heute als "Burger-Virus" den Programmierern bekannt, die Abwehrprogramme schreiben [**?**]. Auch in Deutschland wird nichts gegen derlei Gebaren unternommen. Dadurch entsteht der Eindruck, solche Veröffentlichungen seien in Ordnung. Auf diese Weise wird es sehr schwierig werden, (zukünftigen) Virusprogrammierern den Nachschub an Information zu entziehen.

2.1.2 Verbreitungswege

Wie kommt das Virus ins System? Betrachten wie einen APC, der nichts als ein Netzteil, die Mutterplatine mit Speicher, den Prozessor und eine Graphikkarte enthält und in den kein Kabel eingesteckt ist. Das Netzkabel und die Verbindung zum Monitor sind ein Muß. Wir machen nun ein Gedankenexperiment: Wir stellen alle notwendigen und möglichen Verbindungen her und beobachten die Auswirkungen unserer "Aktionen".

Abbildung 2.1: Wie kommt das Virus ins System?

Tastatur. Mit dem Anschluß der Tastatur ergibt sich die erste und einfachste Möglichkeit zur Vireneingabe, die sich — wenn überhaupt — am schwersten verhindern läßt. Falls das System zur Programmentwicklung vorgesehen ist, kann nichts gegen die Eingabe und Erstellung eines einsatzfähigen Virus, der ja auch nur ein Programm ist, unternommen werden. Zur Abwehr von Softwareanomalien müßte der Compiler in der Lage sein, den Zweck des von ihm bearbeiteten Quelltexts zu erkennen und entsprechende Maßnahmen zu ergreifen. Das dies ein unlösbares Problem ist, zeigt der Beweis in Abschnitt 2.4 "Cohen's Theorien".

Da Programmierer meist auch über weitgehende Systemrechte verfügen, weil diese zur Erfüllung ihrer Aufgaben erforderlich sind, tut sich hier zwangsläufig eine Sicherheitslücke auf, die nicht ohne weiteres zu schließen ist. Neben der sorgfältigen Personalauswahl mit dem Ziel, möglichst vertrauenswürdige Mitarbeiter zu beschäftigen [24], kann mit Hilfe von Logdateien eine gewisse Kontrolle erreicht werden (s.a. Abschnitt 2.7.8 "Vergleich der Konzepte"). Software zur Programmerstellung stellt immer ein großes potentielles Sicherheitsrisiko dar. Falls die Nutzung eines Computers nicht die Erstellung von Programmen beinhaltet, sollten sich keine Übersetzungsprogramme auf dem System befinden.

Floppy Disk/Festplatte. Disketten und — seltener — Wechselplatten sind wahrscheinlich der "Hauptvektor" der Softwareanomalien. Ist die Festplatte fest installiert, kann sie lediglich als Reservoir für infizierte Software dienen und ist nur indirekt an der Verbreitung beteiligt. Speziell Disketten bergen die Gefahr einer Reinfektion nach erfolgreicher Säuberung des Systems, da sie bei dieser Prozedur oft übersehen werden. Befinden sich auf diesen Datenträgern verseuchte Programme, ist bei deren nächsten Benutzung die Entseuchung des Systems oft schon etwas in Vergessenheit geraten und der Umgang damit entsprechend sorglos.

Serielle/parallele Schnittstelle. Für die Datenfernübertragung (DFÜ) gilt im Prinzip das Gleiche wie für Disketten, nur das der Vektor jetzt immaterieller Natur ist. Datenaustausch per DFÜ stellt die potentiell größte Verseuchungsgefahr dar. Der Austausch von Dateien über z.B. ein BBS ist durch folgende Merkmale gekennzeichnet:

- Jeder kann teilnehmen, meistens ohne daß eine beweiskräftige Authentifikation

erfolgt.

- Jeder kann Programme vom BBS laden (*Download*).
- Oft besteht die Möglichkeit, Programme in das BBS zu laden (*Upload*).
- Jeder mit der nötigen Hard- und Software kann BBS-Betreiber werden.

Tatsächlich tragen BBS kaum zur Verbreitung von Softwareanomalien bei, weil die meisten Betreiber die auf dem BBS verfügbare Software sorgfältig prüfen (bis zum nächsten, noch unbekannten Virus...). Ein ganz anderes Kapitel sind BBS, deren Ziel es ist, Informationen über die Programmierung von Computerviren oder sogar Virenprogramme selbst zu vertreiben. Durch den unbeschränkten Zugriff und mangelnde Kontrolle der BBS ist schnell und auf breiter Basis großer Schaden, sprich: Verbreitung der Information angerichtet. Auch Angriffe von Anwendern auf BBS wurden bekannt. In einem konkreten Fall konnte ein Benutzer ermittelt werden, der wiederholt versuchte, ein verseuchtes Programm in ein BBS zu laden. Die Betreiberfirma erstattete daraufhin Anzeige [92].

2.1.3 Motive der Anwender

Den meisten Benutzern von Computersystemen kann nicht unterstellt werden, daß sie Computer wissentlich oder absichtlich infizieren. Was gibt es also an Ursachen dafür, daß Rechner immer wieder mit Viren infiziert werden? Die folgende Aufteilung der Motive orientiert sich am Rechnerbetrieb in einem Betrieb oder Rechenzentrum z.B. einer Hochschule, das den Benutzern Arbeitsplatzrechner vernetzt oder im Stand Alone-Betrieb zur Verfügung stellt.

Gruppe 1. Die meisten Rechner werden wohl durch *Unwissenheit* verseucht und bleiben dies auch. Wer um die Bedrohung durch Computerviren nur etwas aus der Zeitung weiß, ist nicht genügend für die Arbeit mit und an öffentlich zugänglichen Rechnern gerüstet. Das gilt auch für den eigenen PC zu Hause, der eine permanente Virenquelle darstellen kann, gegen die z.B. der Betreiber eines vom diesem Anwender ebenfalls genutzten Rechenzentrums wenig ausrichten kann. Auch aus *Unterschätzung* werden viele Rechner Opfer von Viren. Die meisten Computeranwender, die über Grundkenntnisse auf diesem Themengebiet verfügen, unterschätzen dennoch die Fähigkeiten von manchen Computerviren, Programme zu verseuchen und Infektionen lange Zeit erfolgreich vor dem Benutzer zu verbergen (bes. Stealth Viruses wie “4096” , “Whale” etc. [38] 3.156, 3.357, 3.358).

Selbst diejenigen, die sich mit Computerviren aktiv beschäftigen, gefährden u.U. sich und andere durch ihre *Unachtsamkeit* und *Fahrlässigkeit*. Sie wissen zwar, daß z.B. laut Benutzerordnung keine Programme von Diskette gestartet werden dürfen, tun dies aber trotzdem, weil sie ein Programm schon kompiliert vorliegen haben und im ersten Moment der Start des Kompilats bequemer erscheint.

Benutzer der Gruppe 1 brauchen quasi nur eine “Gedächtnisstütze”, die sie an die Regeln der Benutzerordnung erinnert.

Gruppe 2. Es gibt auch Benutzer, die wissentlich Rechner des Betriebs oder Rechenzentrums der Gefahr der Infektion aussetzen, indem sie fremde Programme mitbringen und diese starten. Grund dafür ist z.B., daß Programme als notwendig erachtet werden, die aber nicht auf dem System gespeichert sind. Oft werden vertraute Hilfsprogramme und Spiele mit an den Arbeitsplatz gebracht, dort benutzt oder mit Kollegen getauscht. Mit nach Hause genommene, auf dem eigenen Rechner bearbeitete und in die Firma zurückgebrachte Arbeit ist ebenfalls eine potentielle Infektionsquelle.

Anwender dieser Gruppe würden wahrscheinlich immer wieder Programme mitbringen, wenn sie niemand daran hindert. Sie sehen aber durchaus den Einsatz von Verboten, Schutzprogrammen und die Gründe dafür ein. Ein Gegenmaßnahme wäre die Einrichtung einer Softwarezulassungsstelle, die mitgebrachte Software überprüft und entweder abweist oder für den regulären Einsatz am Arbeitsplatz freigibt [95].

Gruppe 3. Die dritte Gruppe umfaßt Benutzer, die absichtlich Rechner infizieren oder sogar selbst Viren schreiben. Hierbei kann man ohne weiteres von Computer-Sabotage sprechen; ein Tatbestand, der seit 1986 nach §303 B StGB auch in Deutschland u.U. rechtliche Konsequenzen hat (s.a. Abschnitt 1.6). Diese Anwender sehen Schutzmaßnahmen höchstens als Herausforderung an ihre Fähigkeiten an, in Rechnersysteme einzubrechen.

Im Bereich der wahrscheinlich meist jungen Virenprogrammierer dürften Experimentiertrieb und Selbstbestätigung die stärksten Motive sein, wie sich an typischen Nachrichten zeigt, die man in Viren findet oder die diese ausgeben. Rache ist häufig das Motiv von Mitarbeitern, die sich von ihrer Arbeitsstelle nicht freiwillig getrennt haben. So wird z.B. als “Abschiedsgruß” ein Trojaner oder Virus im System hinterlassen, der sich bei einer bestimmten Gelegenheit aktiviert. Als Gegenmaßnahme sollte durch Änderung oder Löschung des Paßworts die betreffende Benutzerkennung (Account) sofort gesperrt werden. Alle Dateien, zu denen der Mitarbeiter Schreibzugriff hatte, sind zu überprüfen oder, falls möglich, aus dem System zu entfernen.

Vergleichsweise selten aufgetreten sind bisher Viren mit politischen Botschaften. Manche Viren wie “Israeli” weisen durch das Auslösedatum auf ein bestimmtes Ereignis hin, andere wie “Saddam” und “Iraqi Warrior” verbreiten konkrete Nachrichten in Textform.

2.2 Konventionelle Sicherheitsmaßnahmen

2.2.1 Organisation (Kontrollmaßnahmen)

Dieser Abschnitt führt Aspekte der Datensicherheit an, wie sie das BDSG (Bundesdatenschutzgesetz) in der Anlage zu §6 Abs. 1 Satz 1 definiert [94]. Da die Gewährleistung der in der Anlage genannten Anforderungen gemäß §6 BDSG Pflicht für Personen oder Stellen sind, die personenbezogene Daten verarbeiten, existiert wahrscheinlich bereits entsprechende Sicherheitssoft- und Hardware. Diese ließe sich evtl. zum Schutz gegen Computerviren verwenden. Der folgende Text geht die Anforderungen des BDSG Schritt

für Schritt durch und untersucht deren mögliche Eignung für die Abwehr von Softwareanomalien bzw. potentielle Lücken. Die Vorschläge zur Realisierung der Maßnahmen wurden aus [70] entnommen. Der Begriff "sensitive Daten" ist für unsere Betrachtung durch den Begriff "infizierbare Daten" zu ersetzen.

Zugangskontrolle. Nur Befugte haben Zutritt zu Datenverarbeitungsanlagen, mit denen sensitive Daten verarbeitet werden.

- Berechtigte Benutzer festlegen

- Räume und Datenträger abschließen bzw. wegschließen (konventionelle Schlüssel, Ausweisleser, Chipkarten; Closed Shop-Betrieb)

- Geräte durch Hardwaremaßnahmen abschließbar machen
- Überwachungs- und Alarmanlagen vorsehen

Die Zugangskontrolle befaßt sich mit dem physischen Zugang zu Rechneranlagen und Datenträgern; eine Maßnahme, die dem PC-Benutzer zu Hause nur insofern vertraut ist, als daß er sein Haus abschließt, wenn er dieses verläßt. Zugangskontrolle ist nur dann realisierbar, wenn Rechenzentrum und Rechner entsprechend ausgerüstet sind und der Zweck des Rechenzentrumsbetriebs dies zuläßt. Ersteres ist bei konventionellen PCs nicht der Fall, die mit entsprechender Hardware auszurüsten sind. Zweiteres ist bei Universitätsrechenzentren u.ä. nicht gegeben, weil eine wirksame Personenkontrolle zu aufwendig ist und der Grad der Sicherheitsanforderungen meist nicht die Anschaffung teurer Hardware rechtfertigt oder notwendig macht. Natürlich ist es weiterhin möglich, daß eine berechtigte Person unabsichtlich oder willentlich einen Virus auf den Rechner bringt. Mit diesem Punkt beschäftigt sich die Zugriffs- und Eingabekontrolle.

Abgangskontrolle. Nur Befugte können Datenträger entfernen.

- Kataster der mobilen Datenträger anlegen (Disketten, Wechselplatten, portable Computer)
- Datenträgeraustausch protokollieren
- Aufbewahrungsregeln und -fristen festsetzen
- Nicht mehr benötigte Datenträger (auch Druckausgaben) löschen oder vernichten

Die Abgangskontrolle hat die Aufgabe eines Kopier- oder besser Diebstahlschutzes für ganze Datenträger und damit wenig mit dem Schutz vor Computerviren zu tun. Allerdings läßt sich mit Abgangslisten evtl. noch feststellen, an welche Stellen versehentlich verseuchte Software verschickt worden ist. Systemverantwortliche haben so die Möglichkeit, gezielt Warnungen zu verschicken. Ähnliches gilt für die Übermittlungskontrolle (s. dort).

Speicherkontrolle. Nur Befugte können sensitive Daten eingeben, zur Kenntnis nehmen oder verändern (löschen).

- Festlegung der Benutzer und Rechte
- Authentifikation mit Benutzerkennung und Paßwort
- Sperrung oder Entfernung der Diskettenlaufwerke, Verwendung sicherer Fileserver
- Integritätsschutz (Verschlüsselung, Signierung)
- Beschränkung auf notwendige Programme
- Logging und Reporting

Hier ergibt sich prinzipiell das gleiche Problem wie bei der Zugangskontrolle, nämlich daß die Beschränkung des Benutzerkreises nicht garantiert, daß Befugte nicht auch

Schaden anrichten können. Trotzdem werden hier vier ganz wichtige Punkte angesprochen, um die wir uns später noch kümmern werden: Verwendung von Fileservern, Integritätsschutz von Dateien, Beschränkung der Arbeitsumgebung und Protokollierung von Operationen.

Benutzerkontrolle. Nur Befugte können DV-Systeme zur Bearbeitung sensitiver Daten benutzen.

- Zugriffsrechte festlegen und protokollieren
- Sicherheitssoft- und Hardware vorsehen
- Verfahren zur Authentifikation implementieren (Paßwörter, Chipcards)
- Unberechtigte Zugriffe abweisen, protokollieren und Reports ausgeben (Logging)
- Verschlüsselungsverfahren einsetzen (z.B. gegen Diebstahl des Gerätes/Datenträgers)

Die Benutzerkontrolle stellt nach der Zugangskontrolle das zweite Hemmnis auf dem Weg zu den zu schützenden Daten dar. Der Anwender, der physischen Zugriff auf den Rechner hat, muß sich, um Zugang zu Dienstleistungsfunktionen zu erhalten, dem Sicherheitssystem gegenüber identifizieren. Auch hier gilt wieder, daß zulässige Benutzer kein Garant für den Schutz vor Softwareanomalien sind. Um selbst dem Fall vorzubeugen, daß der Rechner oder Teile davon (z.B. die Festplatte) entwendet und die Schutzhardware ausgebaut wird, sind alle Daten zu verschlüsseln.

Zugriffskontrolle. Jede Person kann nur auf sensitive Daten innerhalb ihrer Zugriffsberechtigung zugreifen.

- APCs in LANs eindeutig identifizieren
- Ressourcen einschränken (Zugriffszeiten, Sperren der Betriebssystemebene, zulässige Betriebssystemfunktionen, zulässige Betriebsmittel)
- s.a. Speicher- und Benutzerkontrolle

Die Zugriffskontrolle ist ein ganz wesentlicher Punkt für den Schutz vor Softwareanomalien. Hier geht es um den Transport von Daten auf den Rechner und damit um die Einschleusung potentiell gefährlicher Programme.

Übermittlungskontrolle. Die Stellen, an die sensitive Daten übermittelt werden können, sind überprüft und festgestellt.

- Netzwerkdokumentation
- Protokollierung der DFÜ (DFÜ-Programme, Sender, Daten, Empfänger)

Ziel der Übermittlungskontrolle ist es, den Überblick über mögliche Datenquellen und -senken zu behalten und den Datenaustausch mit diesen zu protokollieren. Zusammen mit der Abgangs- und der Eingabekontrolle realisiert die Übermittlungskontrolle die vollständige Protokollierung aller Datenbewegungen zwischen Rechner (Rechenzentrum) und Außenwelt. Auf diese Weise lassen sich Gefahrenquellen prophylaktisch erkennen und Infektionswege im Nachhinein verfolgen.

Eingabekontrolle. Es kann nachträglich festgestellt werden, wer wann welche sensitive Daten eingegeben hat.

- Protokollierung der Zugriffsrechte
- Logging (Verarbeitung "Wer – Wann – Was", Transaktionen)

Ähnlich der Abgangs- und Übermittlungskontrolle läßt sich evtl. über Auswertung der Protokolle "post mortem" der Verursacher oder Weg einer Verseuchung feststellen, wenn sie schon nicht verhindert werden konnte.

Auftragskontrolle. Daten können nur dem Auftrag entsprechend verarbeitet werden.

- Vertrag eindeutig gestalten
- Auftragnehmer sorgfältig auswählen
- Vertragsausführung kontrollieren (evtl. Vertragsstrafen bei Verletzung)

Die Auftragskontrolle soll sicherstellen, daß bei der Verarbeitung sensitiver Daten durch Dritte nur die gewünschten Aufgaben durchgeführt und die Daten nicht anderweitig genutzt werden. Der Auftragnehmer kann die übergebenen Daten möglicherweise manipulieren. Mit diesem Aspekt wollen wir uns bei der Transportkontrolle beschäftigen.

Transportkontrolle. Sensitive Daten können beim Transport weder gelesen noch verändert (gelöscht) werden.

- Daten verschlüsseln (Sicherung von Vertraulichkeit, Integrität und Authentizität)
- Daten kopierschützen
- Auf Vollständigkeit überprüfen
- Transportmodalitäten festlegen (Verpackungs- und Versandvorschriften, Transportwege, Versandart wie Kurierdienste, Einschreiben etc.)

Auf Softwareanomalien übertragen bedeutet Transportkontrolle, daß eine Manipulation der Software auf dem Weg zwischen zwei Stellen ausgeschlossen ist. Dies ist nicht so einfach zu realisieren: Einer Diskette sieht man nicht an, ob sie evtl. zu Vorführungszwecken beim Händler eingesetzt worden ist oder von einem Kunden nach Ablauf der Testperiode zurückgegeben wurde. Abhilfe ist durch versiegelte Verpackungen (z.B. Einschweißen in Plastikhülle) oder nicht beschreibbare Disketten (keine Schreibschutzkerbe) möglich [38] 3.011ff.

Organisationskontrolle. Die Organisation wird den Anforderungen des Datenschutzes gerecht.

- Funktionen möglichst trennen
- Regeln für Benutzer, Programmierung, Dokumentation, Test, Freigabe, Bedienung, Verarbeitung, Aufbewahrung und Entsorgung aufstellen
- Benutzerrechte revisionsfähig dokumentieren

- Einheitliche Verfahren zur Beschaffung von Hard- und Software einführen
- Kontrollen auf Einhaltung der Regeln durchführen
- Hard- und Softwarekataster anlegen; Verfahren zu Inventur festlegen
- Entsprechende Versicherungen abschließen

Die Organisationskontrolle ist neben der Zugriffskontrolle ein weiterer, wesentlicher Punkt bei der Bekämpfung von Softwareanomalien. Zur wirksamen Abwehr gehört neben dem Einsatz von Sicherheitssoft- und Hardware die Einrichtung von Stellen in der Organisation, die deren Beschaffung und Einsatz überwachen und die Einhaltung von Vorschriften stichprobenartig und ohne Vorankündigung vor Ort kontrollieren [95].

2.2.2 Grundlegende Maßnahmen

Saubere Rechner. *Saubere*, d.h. virenfreie Rechner sind die Basis aller Maßnahmen gegen Computerviren. Zunächst wird der Rechner mindestens eine Minute lang ausgeschaltet, um den flüchtigen Speicher (RAM) und damit alle in ihm gespeicherten (Viren-) Programme sicher zu löschen. Dies ist wichtig, weil manche Viren durch einen Reset nicht entfernt werden können. Für alle folgenden Tätigkeiten sollte man nur mit schreibgeschützten Originaldisketten arbeiten, damit weder der Rechner noch die Originale verseucht werden können. Als nächster Schritt wird das Betriebssystem urgeladen und die Festplatte evtl. formatiert[2]. Das Betriebssystem und alle anderen Programme werden auf einer später möglichst schreibgeschützten Partition (logisches Laufwerk) installiert. Einen solchen softwaremäßigen Schutz implementiert z.B. der Gerätetreiber `dmdriver.sys` aus dem Paket "Disk Manager" der Firma Seagate oder unser noch zu entwickelnder Watcher `AVWatchP`.

Saubere Software. Die Verwendung von Original-Software ist kein Garant dafür, daß die Programme virenfrei sind. In der Vergangenheit kam es zu einigen Zwischenfällen mit infizierten Programmen und Disketten, die schon ab Hersteller mit einem Virus behaftet waren. So verschickte Anfang 1990 das amerikanische Volkszählungsbüro irrtümlich mit dem "Jerusalem"-Virus verseuchte Disketten, auf denen sich Retrieval-Software für Daten auf CD-ROM befand [38] 3.080. Ebenfalls Januar 1990 verschickte die RWTH Aachen Disketten mit dem "Cascade"-Virus an Hochschulen und Rechenzentren in Nordrhein-Westfalen. Virusträger war ein Programm namens `arcx.com`, das zu einem elektronischen Fragebogen gehörte. Die Verantwortlichen wollten sich zum Vorgang leider nicht äußern; dadurch blieb die Infektionsursache im Dunklen [54].

Firmen wie z.B. die DATEV[3] in Nürnberg überprüfen jedes Fremdprogramm ausgiebig auf bekannte Viren und führen Tests in einer isolierten Umgebung durch, bevor das Programm für die Benutzung freigegeben wird. Speziell für öffentliche Stellen, militärische Einrichtungen und Firmen kommt die Möglichkeit in Betracht, daß jemand absichtlich versucht, ein Virus einzuschleusen. Sabotage, Terrorismus, Spionage und

[2] `format` reicht für manche BSIs auf Festplatte *nicht* aus; `fdisk` benutzen!
[3] Datenverarbeitungsorganisation des steuerberatenden Berufs in der BR Deutschland e.V.

politische Ambitionen sind als Motive für Viren vielleicht nicht so utopisch, wie sie manchem erscheinen mögen.

Möglichkeiten beschränken (Need to Have-Prinzip). Falls daran gedacht wird, Schutzprogramme einzusetzen, dürfen keine "überflüssigen" Softwarewerkzeuge auf dem System gespeichert sein, die Operationen ermöglichen, die für den bestimmungsgemäßen Betrieb nicht notwendig sind (Prinzip "Need to Have" = "nur haben, was man braucht"). Dazu zählen MS-DOS-Kommandos zur Festplattenformatierung (`format`, `fdisk`), Änderung von Dateiattributen (`attrib`) und Datensicherung (`backup`, `restore`). Werkzeuge wie `debug`, `symdeb`, PC-Tools, Norton Utilities und andere umfassen und übertreffen z.T. noch erheblich die Fähigkeiten der DOS-Kommandos. Solche Programme könnten Schutzmaßnahmen ausschalten oder sie umgehen helfen.

Die Einschränkungen sollten aber nicht so weit gehen, daß quasi ein Zwang entsteht, Programme mitzubringen. Hier muß ein Kompromiß zwischen Gefährdung des Rechners und des Schutzprogramms durch installierte Programme und dem empfundenen "Zwang", (verseuchte) Programme mitzubringen, gefunden werden.

Vorhandene Möglichkeiten ausschöpfen. MS-DOS verfügt über keine Schutzmechanismen, wie sie für ein wirklich professionelles Betriebssystem wie z.B. UNIX selbstverständlich sind. UNIX verlangt vom Benutzer eine Identifikation durch Namen und Paßwort. Für den Schutz von einzelnen Dateien und Katalogen werden verschiedene Zugriffsrechte (engl. read, write, execute = Lesen, Schreiben, Ausführen) und Benutzergruppen (engl. user, group, others = Benutzer, Gruppe, Rest) unterschieden. Trotzdem hört man oft, daß in UNIX-Rechner eingebrochen wurde oder das UNIX generell unsicher sei.

Das hängt in den meisten Fällen aber damit zusammen, daß die vorhandenen Schutzmaßnahmen nicht ausgeschöpft wurden. Paradebeispiel sind triviale Paßwörter, die sich mit der meist frei zugänglichen Datei `passwd`[4] und Computerunterstützung leicht raten lassen. Auch bei der Vergabe von Dateirechten (bes. Set UID-Bit) sind Fehler möglich, die Unbefugten die Erlangung des Superuser-Status ermöglichen. Bekannte, aber nicht beseitigte Bugs sind eine weitere Gefahrenquelle. Keine der drei Schwachstellen basiert auf Fehlern des Betriebssystems, sondern auf Unzulänglichkeiten der Systemverwaltung.

Zugriff einschränken (Need to Know-Prinzip). Immerhin unterstützt MS-DOS vier Dateiattribute (Tab. 2.1). Interessant ist vor allem der Schreibschutz, der ohne ein entsprechendes Programm nicht umgehbar ist. Das `readonly`-Bit ist außerdem das einzige Attribut, das mit `attrib` beeinflußbar ist. Dateien mit `hidden`- oder `system`-Status werden beim Auflisten mit `dir` nicht angezeigt. Programme und Dateien, die für den Benutzer nicht zugänglich oder veränderbar sein sollen, können mit entsprechenden Werkzeugen unbeschreibbar und unsichtbar gemacht werden (Prinzip "Need to Know" = "nur wissen, was man braucht").

Werden der externe DOS-Befehl `attrib` und andere Programme mit ähnlichen Fähigkeiten aus dem System entfernt, kann der Benutzer kein Attribut mehr verändern. Wichtige Dateien wie die Programme und Konfigurationsdaten eines Antivirus-Pakets

[4] Enthält die mit `crypt` einwegverschlüsselten Paßwörter.

Attribut	Bedeutung
`archive`	zu archivieren
`readonly`	nur lesen
`hidden`	versteckt
`system`	Systemdatei (versteckt)

Tabelle 2.1: Dateiattribute unter MS-DOS

sind verborgen und schreibgeschützt nicht mehr ohne weiteres manipulierbar, aber weiterhin normal zu benutzen. Ein derartiger Schutz erschwert das Auffinden und die Manipulation sicherheitsrelevanter Daten, auch für manche Viren. Programme wie `fa` (**F**ile **A**ttributes) aus den Norton Utilities sind in der Lage, alle Attribute zu verändern.

2.2.3 Ethik

Bewußtsein für Viren schaffen. Die meisten Anwender wissen von Computerviren aus Rundfunk und Presse, die das Thema zwar publikumswirksam aufbauschen, aber selten nutzbare Informationen vermitteln. K. Brunnstein, einer der führenden Forscher in Sachen Softwareanomalien, bezeichnet den "Virus Desinformaticus" als den schlimmsten seiner Art [26]. So hat z.B. der Autor im Rechenzentrum einer Hochschule immer wieder beobachtet, daß die Benutzer und sogar zuständiges Personal nicht wissen, daß die Rechner regelmäßig verseucht sind. Die Anwender reagieren entweder verschreckt oder gar nicht, wenn man sie auf diese Tatsache hinweist.

Viele Benutzer wissen nicht, welche Folgen eine Infektion des Rechners mit Computerviren haben kann. Diese Gefahren sollten im Rahmen des Informatikunterrichts an der Schule oder des Programmierpraktikums im Grundstudium angesprochen werden. Dadurch könnte erreicht werden, daß neben den Programmierkenntnissen auch ein Gefahren- und Verantwortungsbewußtsein vermittelt wird. Benutzer der Gruppen 1 und 2 kann man wahrscheinlich durch solche Maßnahmen erreichen, Anwender der Gruppe 3 dagegen vermutlich nicht.

Es sollte darauf hingewiesen werden, daß Computerviren kein "Problem anderer Leute" sind, sondern auch der eigene Rechner zu Hause sehr schnell Opfer einer programmierten Attacke werden kann. Die möglichen Konsequenzen: Unwiederbringliche Zerstörung von Daten und Programmen, evtl. Formatierung der Festplatte, Verlust an Zeit und Geld und viele andere unangenehme Dinge mehr. Die Benutzer sehen dann schnell ein, warum ein Rechenzentrum besonders geschützt werden muß.

2.3 MS-DOS und andere Betriebssysteme

Die folgenden beiden Abschnitte behandeln und vergleichen die Sicherheit von MS-DOS und anderen Betriebssystemen. Zu unterscheiden ist zwischen der inneren Sicherheit, die vom Betriebssystem und der verwendeten Hardware geboten wird, und der äußeren

Sicherheit, die von organisatorischen Maßnahmen und von der Art des Rechnerbetriebs abhängt. Für jeden Sicherheitsaspekt wird dessen Bedeutung für den Schutz vor oder die Begünstigung von Softwareanomalien untersucht.

2.3.1 Innere Sicherheit (Zugriffskontrolle)

Für Großrechner konzipierte Betriebssysteme wie UNIX (AT&T), VMS (DEC) und MVS (IBM) verfügen über Zugriffskontrollsysteme auf Datei- und Verzeichnisebene. Durch sie wird festgelegt, welcher Benutzer (Subjekt) welche Operationen mit welchen Objekten durchführen darf. Operation/Objekt-Kombinationen sind z.B. das Lesen einer Datei, Verändern des Datums, Formatieren eines Datenträgers, Umbenennen eines Programms usw.

MS-DOS verfügt wie alle älteren Betriebssysteme für APCs über keinerlei Zugriffskontrollen. Dem Benutzer stehen alle Operationen, die MS-DOS überhaupt anbietet, zur Verfügung. Jeder APC-Benutzer ist, falls keine Sicherungsmaßnahmen getroffen werden, Anwender, Operator und Systemverwalter in einer Person. Angesichts der unter 2.2.1 aufgeführten Anforderungen des BDSG darf demnach mit einem ungeschützten MS-DOS-PC keine Verarbeitung personenbezogener Daten vorgenommen werden. Die Installation von Sicherheits-Hard- und Software ist deshalb ein Muß.

Neben dem Einsatz alternativer Betriebssysteme für APCs wie UNIX besteht die Möglichkeit, MS-DOS-kompatible Betriebssysteme zu verwenden, die über Sicherheitseinrichtungen verfügen. Den ersten Schritt auf diesem Gebiet machte Digital Research 1990 mit seinem DR-DOS 5.0, das Schutz auf Datei- und Verzeichnisebene realisiert.

Lücken in der Zugriffskontrolle. Es lassen sich zwei grundsätzliche Arten von Schwächen in Schutzmaßnahmen unterscheiden. Zum einen existieren Designfehler im Systemkern oder Systemprogrammen, die zu Fehlfunktionen führen. Zum anderen gibt es konzeptionelle Fehler, die ihre Ursache nicht im Programmversagen, sondern in der Spezifikation haben.

Als Beispiel für einen Designfehler diene das UNIX-Systemprogramm `fingerd`, bei dem ein interner Überlauf unerwünschte Effekte hervorrufen konnte (s.a. 1.1 "INTERNET-Worm"). Ein anderer denkbarer Fehler wäre z.B. die Freigabe aller Operationen für hohe Benutzernummern, weil der Kontrollalgorithmus für eine bestimmte maximale Zahl ausgelegt ist. Bei der Übergabe einer zu großen Benutzernummer werden Daten gelesen, die nicht zum gewünschten Anwender oder überhaupt zur Tabelle der Benutzerrechte gehören.

Bei konzeptionellen Fehlern funktionieren z.B. Algorithmen des Schutzprogramms wie vom Ersteller erwartet, nur wurden mögliche Sicherheitslücken übersehen. Dazu zwei Beispiele aus der Praxis:

- Der Window-Manager für DEC-Windows (unter VMS) läuft mit Systemprivilegien ab, konnte aber trotzdem vom Benutzer selbst und beliebig gewählt werden. Das galt speziell auch für eigene Programme des Anwenders [81]

- Mit Hilfe des `Gnu-Emacs`-Editors (unter UNIX) kann eine Datei einem anderen Benutzer zugänglich gemacht werden. Dazu kopiert `Emacs` die Datei an ein bestimmtes Ziel und trägt den Empfänger als neuen Besitzer ein. Damit erbt die Datei zugleich alle Rechte des Adressaten. Weil `Emacs` weder Ziel noch Empfänger (Superuser!) überprüfte, konnten Systemprogramme gegen eigene Produkte ausgetauscht werden [**?**][5].

Derartige Sicherheitslücken sind schwer aufzuspüren und bleiben deshalb evtl. über lange Zeit hinweg unbekannt. Findet ein Hacker sie vor den Systemverantwortlichen heraus, so stehen ihm Tür und Tor offen. Selbst nach Aufdeckung der Sicherheitslücke sind vom Hersteller noch Gegenprogramme zu entwickeln und an jeden Kunden zu verbreiten, der sie hoffentlich auch bald oder überhaupt installiert. Untersuchungen des INTERNET-Worm Zwischenfalls haben gezeigt, daß Abhilfe zwar innerhalb weniger Tage zur Verfügung stand, aber z.T. sogar Monate später noch nicht installiert worden war.

Shells. Nach dem Einloggen (sich mit Benutzernamen und Paßwort anmelden) wird unter UNIX automatisch ein Programm, eine sog. Shell, gestartet, die dem Benutzer elementare Funktionen zur Bearbeitung von Dateien zur Verfügung stellt. Diese Shell kann so gestaltet werden, daß nur bestimmte, für die Aufgaben des Benutzers erforderliche Funktion ausführbar sind.

Isolierte Umgebung. Unter UNIX kann der Systemadministrator mit dem Kommando `chroot` (**c**hange root) ein Verzeichnis festlegen, welches als Wurzelverzeichnis für bestimmte Benutzer dienen soll. Damit kann ein Teilbaum des Dateiverzeichnisses quasi abgetrennt werden; die betreffenden Anwender können den Sub-Baum, der ihnen wie ein komplettes Verzeichnis erscheint, nicht verlassen. Nützlich ist diese Methode z.B. bei der Einrichtung eines abgeschotteten Teilbereichs für Anwender, die über Netze auf den Rechner zugreifen. Zwar kann sich jeder in das System einloggen, ist aber trotzdem gegenüber den lokalen Benutzern in seinen Möglichkeiten eingeschränkt. Fast auf allen UNIX-Rechnern, auf denen FTP für jedermann angeboten wird (Anonymous FTP, s.a. Anhang B), ist diese Sicherheitsmaßnahme gebräuchlich.

Speicherschutz. Auch wenn ein APC unter UNIX oder einem anderen Betriebssystem mit Sicherheitseinrichtungen betrieben wird, ist noch lange nicht die Qualität des Schutzes erreicht, die ein Großrechner erreichen kann. Die Ursache hierfür liegt in der Hardware begründet. APCs verfügen i.d.R. über keine Mechanismen zur hardwaremäßigen Speicherkontrolle. Jedes Programm kann wahlfrei auf jeden Bereich des Speichers lesend und schreibend zugreifen. Eine Ausnahme bilden Rechner mit INTEL-CPUs jenseits des 286ers, die wie ihre großen Kollegen geschützten Speicher (Protected Memory) realisieren können (nicht unter MS-DOS).

In diesem Modus ist der durch ein Programm belegte Speicher in genau definierte Bereiche, sog. Segmente, unterteilt. Ein Programm kann nur in Codesegmenten ablaufen, nur in Datensegmenten ist das Lesen und Schreiben von Daten möglich. Operationen, die diese Regeln verletzen, werden bereits auf Hardwareebene erkannt, abgefangen und der Sachverhalt an das Betriebssystem weitergemeldet. Neben einer entsprechenden

[5] Dieses Verfahren wurde u.a. beim 1989 bekanntgewordenen "KGB-Hack" verwendet.

Meldung für Logbuch und Anwender wird das verantwortliche Programm abgebrochen. Durch diese Maßnahmen können Viren nur auf Programme und Daten zugreifen, die sich nicht im Speicher, sondern auf externen Datenträgern befinden. Darüber hinaus muß das Virusprogramm entsprechende Dateizugriffsrechte besitzen. Das Betriebssystem ist damit gegen Manipulationen geschützt.

2.3.2 Äußere Sicherheit (Zugangskontrolle)

Closed Shop-Betrieb. Im Mini- und Mainframesektor ist es möglich und auch allgemein üblich, die eigentlichen Rechner und Peripheriegeräte wie Platten-, Disketten- und Magnetbandlaufwerke und Drucker in einem besonderen, für normale Benutzer nicht zugänglichen Raum unterzubringen. Im diesem sog. *Closed Shop*-Betrieb fordern Benutzer von einem zentralen Rechenzentrum Dienste an. Die Zentralisierung ermöglicht die einfache und wirkungsvolle Installierung von Schutzmechanismen.

Stand Alone-APCs. Mit dem Aufkommen von APCs entstand ein bis heute anhaltender Trend zur dezentralen Datenverarbeitung, der einige Schwierigkeiten im Hinblick auf Schutzmaßnahmen mit sich bringt. Die große Anzahl der Rechner und ihrer Standorte stellen hier das Hauptproblem dar.

Vernetzte APCs. APCs sind häufig über LANs (**l**ocal **a**rea **n**etworks = lokale Netzwerke) miteinander vernetzt, die eine potentielle Expreßstrecke für die Verbreitung von Softwareanomalien darstellen. Außerdem bringt die Übertragung von Daten auch Risiken wie z.B. das Abhören oder die Auswertung kompromittierender Abstrahlung mit sich. Diese Gefahren können durch die Verschlüsselung der zu übertragenden Daten und Verwendung von Glasfaserkabeln[6] auf ein Minimum reduziert werden.

Der Einsatz eines zentralen Fileservers hat andererseits Vorteile, wie sie ein Rechenzentrum bietet [72]. Der Fileserver, über den die angeschlossenen APCs die benötigten Programme, Daten und vor allen Dingen das Betriebssystem laden[7], kann in einer kontrollierten Umgebung aufgestellt und mit Sicherheitssoft- und Hardware ausgerüstet werden. Einige Hersteller bieten spezielle Server-Rechner an, bei denen alle Bedienungselemente und Öffnungen für Schrauben hinter einer abschließbaren Blende liegen. Datensicherung und -schutz sowie Überwachung ist zentral über die Netzwerksoftware möglich, die analog zu Großrechnerbetriebssystemen Schutz auf Datei und Verzeichnisebene bietet und den Einsatz von Programmen überwacht und kontrolliert.

Falls kein lokaler Datenaustausch über Datenträger (Disketten) notwendig ist, kann auf Diskettenlaufwerke verzichtet werden ("Diskless Workstation"). Durch ein spezielles BIOS, das meist der Netzwerkhersteller liefert, wird das Betriebssystem nicht mehr von Diskette, sondern nur noch vom Fileserver geladen. Es können weder verseuchte Programme eingespielt noch auf dem lokalen APC oder dem Fileserver gespeicherte Daten illegal kopiert werden.

Neben sicherheitstechnischen Vorteilen bieten Server-Konzepte auch wirtschaftliche Vorteile: Sicherheitseinrichtungen, Programme und Daten, Anschlüsse an andere

[6] Bei *direktem* Zugang zum Kabel können Abstrahlungseffekte genutzt werden.
[7] Falls das BIOS entsprechend modifiziert wird.

Netze (TELEX, TELEFAX etc.), best. Ressourcen wie Plotter, Laserdrucker etc. müssen nur einmal vorhanden sein; neue Dienste wie Electronic Mail kommen quasi als Bonus hinzu. Das, was die Netzkarte für die einzelnen APCs und die Verkabelung kostet, kann zu einem guten Teil wieder damit hereingeholt werden, daß keine lokalen Festplatten und evtl. Diskettenlaufwerke mehr benötigt werden.

Das Novell-Virus. Fileserver-Konzepte bieten natürlich nur dann erhöhte Sicherheit gegenüber Stand Alone-APCs solange der Server selbst nicht verseucht wird. Ende 1990 wurde von Jon David in Zusammenarbeit mit der Firma Novell ein Virus erforscht, das in der Lage ist, Schutzeinrichtungen der Server-Software zu umgehen [49]. Die "Zusammenarbeit" bestand darin, daß sich Novell einen Tag vor dem Auslösedatum des Virus nach langen Verhandlungen dazu bereit erklärte, Versuche auf verschiedenen Netzkonfigurationen durchzuführen. Obwohl die Demonstration erfolgreich verlief, reagierte Novell aggressiv auf eine Mitteilung Jon Davids an die Presse. Diese Art von Zusammenarbeit schadet sowohl den Bemühungen zur Bekämpfung von Computerviren als auch dem Ansehen der betroffenen Firma.

Wo sind die Mainframe-Viren? Diese Frage wurde auf der Diskussionsliste `VIRUS-L` (s. Anhang B) oft gestellt und ebensooft kontrovers diskutiert. Tatsache ist, daß Viren auf Großrechnern ohne weiteres mit geringem Aufwand realisierbar, aber in "freier Wildbahn" praktisch noch nie aufgetreten sind; zumindest wurden keine Fälle publik. Auf APCs unterschiedlichster Hersteller dagegen (IBM, Apple, Atari, Commodore etc.) hat die Zahl der Virenfamilien und ihrer Abkömmlinge 1991 die 1000er-Marke überschritten.

Wie lauten nun die Erklärungen für den sicher nicht beklagten Mangel an Viren auf Großrechnern?

- Die Mainframe-Population ist verglichen mit der Anzahl der APCs sehr gering. Viele Rechner heißt auch viele Programmierer unterschiedlichster Motivation und damit ein guter Nährboden für die Ausbreitung.
- APC-Benutzer tauschen oft Programme miteinander aus, wozu zumeist Disketten (Bootviren!) benutzt werden. Diese Software ist oft auf unüberschaubaren Kanälen weit gereist. Betreiber von Mainframes kaufen Originalsoftware ab Hersteller oder lassen Programme im Haus erstellen und beschränken den Informationsaustausch auf reine Datentransfers.
- angestellte Programmierer haben einen Job zu verlieren und sind, falls das Virus in der eigenen Firma in die Welt gesetzt wird, leicht zu ermitteln.
- Berufsprogrammierer dürften nicht zuletzt auf Grund ihres Alters ein anderes Gefühl für Verantwortung haben, als die meist jungen Virenprogrammierer (ethische Aspekte, "Berufsethos").
- alle IBM-kompatiblen PCs sind auch binärkompatibel, d.h. Programme laufen auf jedem Rechner, weil jede CPU die gleiche Maschinensprache spricht. In der UNIX-Welt ist zwar Kompatibilität auf Quelltextebene einigermaßen gegeben; die verschiedenen Rechnermodelle jedoch verwenden zumeist unterschiedliche CPUs. Dadurch ist eine Infektion von Großrechnern unterschiedlicher Hersteller durch

dasselbe Virus ebensowenig möglich wie die Verseuchung eines Apple Macintoshs durch ein MS-DOS-Virus.

- Mainframes sind von der Hard- und Software her viel komplizierter als APCs aufgebaut; zudem sind Informationen über Hardware und Betriebssysteme nicht jedermann zugänglich. Diese Tatsachen setzen beim Programmierer, der Viren entwickeln und plazieren will, mehr als das übliche Fachwissen und Können voraus.
- Sicherheitseinrichtungen auf Organisations-, Hard- und Softwareebene verhindern die unbefugte Benutzung des Rechners, das Einspielen unerwünschter Programme und die unkontrollierte Ausbreitung von Softwareanomalien.
- durch Logdateien können illegale oder verdächtige Operationen und ihre Verursacher zumindest nachträglich aufgespürt werden (Gefahr des Entdecktwerdens).

Es sprechen aber auch einige Anzeichen dafür, daß die momentane Situation vielleicht nur eine glückliche Fügung ist und die Schwierigkeiten noch bevorstehen:

- wie Cohen in seinen Experimenten zeigte, sind Sicherheitseinrichtungen in Bezug auf Computerviren schlicht unwirksam, weil diese mit den legitimen Rechten der Systembenutzer arbeiten. Intime Kenntnisse des Betriebssystems und evtl. vorhandener Schlupflöcher sind daher nicht notwendig, können aber ein Virus u.U. effektiver machen.
- alle Betriebssysteme haben Schwachstellen, die von Sachkundigen ausgenutzt werden könnten. Von z.B. UNIX ist der Quellcode erhältlich, anhand dessen sich bestimmte Funktionen gezielt auf Schwachstellen untersuchen lassen (z.B. der `fingerd`-Bug).
- bestehende Schutzmöglichkeiten werden oft nur unzureichend genutzt (zu großzügig vergebene Rechte, zu lasche Voreinstellungen für Weglaßwerte, nicht veränderte Paßwörter des Herstellers, triviale Paßwörter etc.).
- einige Hersteller streben die Binärkompatibilität ihrer Systeme an. Damit könnten Programme einfach zwischen Systemen verschiedener Herkunft ausgetauscht werden; die Ausbreitung von Viren wird begünstigt.
- Quelltext-Viren können durch Hardware auferlegte Grenzen überwinden und sind nur noch vom Betriebssystem abhängig. Bereits 1983 zeigte Ken Thompson, einer der Entwickler von UNIX, wie der "C"-Compiler des Systems geeignet manipuliert werden kann (s. 5.2.3 "... und der ganze Rest").
- vielleicht wurden in der Vergangenheit erfolgreiche Mainframe-Viren nur Verschwiegen — Hersteller und Anwender, meist große Firmen, haben einen Ruf zu verlieren.

Fazit: Sowohl Mainframes, Micros als auch APCs bis zu den Homecomputern herunter sind für Computerviren anfällig, wenn auch in unterschiedlicher Weise. Cohens in Experimenten bestätigte Theorien besagen, daß sich ein Virus nur der normalen

Rechte der Benutzer eines Systems bedienen muß, um sich erfolgreich zu verbreiten. Konventionelle Schutzsysteme greifen daher bei dieser Art des elektronischen Vandalismus nicht, sondern erschweren bestenfalls die rasche Ausbreitung.

Computer ohne Schutz auf Hard- und Softwareebene erleichtern ebenso wie Schwachstellen in den vorhandenen Sicherheitssystemen die Arbeit der Computerviren. Die "großen" Computer sind gegenüber den "kleinen" nicht prinzipiell im Vorteil. Abhilfe könnten zukünftige Normen und Prüfverfahren schaffen, wie sie das "Orange Book" und die IT-Sicherheitskriterien vorsehen. Einzig der Schutz der Datenintegrität im Zusammenhang mit sicheren Betriebssystemen kann Softwareanomalien Paroli bieten.

2.4 Konzepte zur Virenabwehr (Cohen's Theorien)

Fred Cohen beschäftigte sich in seinen Arbeiten nicht nur mit der Theorie der Computerviren, sondern entwickelte auch Methoden zur ihrer Abwehr. Insgesamt veröffentlichte Cohen drei Aufsätze u.a. in der Zeitschrift "Computers and Security". Der erste Artikel, "Computer Viruses: Theory and Experiments" [21], wurde bereits in Kapitel 1 angesprochen, soweit er Informationen zur prinzipiellen Funktion und Experimenten mit Computerviren betrifft. Der zweite Teil des Aufsatzes befaßt sich mit der Funktion und Wirksamkeit gängiger Sicherheitskonzepte sowie Untersuchungen zur Detektierbarkeit und Abwehr von Viren. Dazu klärt Cohen zunächst folgende Begriffe:

- *Informationsaustausch* (oder Sharing von engl. to share = teilhaben). Information muß, wenn sie nützlich sein soll, auf Transportkanälen ausgetauscht werden können. Ein System, das keine Informationen mit seiner Umgebung austauscht, heißt *isoliert*.
- *Transitivität*. Wenn eine Instanz (Benutzer oder Programm) A Daten mit B austauschen kann und B mit C, dann kann auch A mit C kommunizieren.
- *Allgemeinheit der Interpretation*. Allgemein ist keine Unterscheidung zwischen Daten und Programmen möglich, da Daten potentiell durch ein Programm interpretiert werden können.

Daraus ergeben sich eine Reihe von Implikationen:

- Wenn Informationen interpretiert (gelesen, bearbeitet und geschrieben) werden, ist eine Infektion möglich.
- Wo Information ausgetauscht wird, können sich Computerviren verbreiten.
- Falls es keine Beschränkung des Informationsflusses gibt, kann eine Infektion von jeder Quelle aus jedes Ziel erreichen.

Auf dieser Grundlage zeigt Cohen auf, daß gängige Schutzsysteme nicht in der Lage sind, Infektionen durch Computerviren generell zu verhindern. Doch auch bei der

gezielten Abwehr von Computerviren ergeben sich Schwierigkeiten, wie die folgenden Beweise zeigen.

Unmöglichkeit eines universellen Virus Detektors (UVD).
Beweis: Es existiere eine Funktion D, die über eine Datei eine Aussage macht, ob sie ein Virus enthält (nach Cohen: ob sie ein Virus ist).
Es existiere ein Programm `virus`, das die Funktion D auf sich selbst anwendet und auf das Ergebnis folgendermaßen reagiert:
D: `virus` ist ein Virus → `virus`: infiziert keine Programme → `virus` ist doch kein Virus → Widerspruch
D: `virus` ist kein Virus → `virus`: infiziert Programme → `virus` ist doch ein Virus → Widerspruch

Unmöglichkeit, die Mutation eines Virus zu entdecken.
Die Beweisführung erfolgt analog: Die Funktion D mache eine Aussage über die funktionale Äquivalenz zweier Programme. Die Programme $\texttt{virus}_1$ und $\texttt{virus}_2$ benutzen D um zu entscheiden, ob sie gleichartig oder verschieden reagieren.

Unmöglichkeit, ein Virus an seinem Verhalten zu erkennen.
Der Beweis lehnt sich an den ersten Beweis an: Jeder Compiler, also ein legitimes Programm, kann bei entsprechender Eingabe ein verseuchtes Programm erzeugen. Um automatisch bestimmen zu können, ob sich ein Compiler wie ein Virus verhält, d.h. wie ein Programm, das Viren produziert, muß eine Funktion D eine Aussage darüber treffen können, ob die Eingabedaten ein Virus erzeugen. Dies ist, wie oben gezeigt, unmöglich.

Cohen zieht aus seinen Untersuchungen folgenden Schluß: Die *sichere* Detektion eines Virus im *Voraus* ist unmöglich. Analoges gilt auch für alle anderen Arten von Softwareanomalien oder allgemeiner die Funktion jedes beliebigen Programms. Trotz dieser vielleicht etwas entmutigenden Erkenntnisse ist die Detektion bekannter Viren (Scanning), virentypischer, sicherheitsgefährdender Operationen (Watching) oder von Veränderungen, die Konsequenz ihrer Funktionsweise sind (Checking), ohne weiteres möglich. Diese drei und andere Methoden werden im unmittelbar folgenden Abschnitt behandelt.

Cohen führt eine weitere, schon recht spezielle Methode an: die *Immunisierung* von Programmen. Die meisten Viren versuchen aus Gründen der Unauffälligkeit, ein Programm nicht mehrfach zu infizieren, da dieses immer größer würde und jeder Infektionsprozeß Zeit benötigt. Deshalb untersucht das Virus das potentielle Opfer auf ein bestimmtes Merkmal, das es bei der Infektion installiert. Durch Nachbildung dieses Merkmals kann das Virus getäuscht und das Programm quasi geimpft werden. Angesichts der Fülle von Viren und ihrer Funktionsweisen ist es jedoch unmöglich, das Infektionsmerkmal jedes Virus anzubringen. Des weiteren umgehen clever programmierte Softwareanomalien wie der "INTERNET-Worm" (s. dort) solche Maßnahmen, indem das Infektionsmerkmal nach einem bestimmten Algorithmus ignoriert wird.

2.5 Kommerziell verfügbare Konzepte

Dieser Abschnitt stellt Verfahren zur Virenabwehr und dazu exemplarisch Programme vor, die kommerziell oder als Share- und Freeware auf dem Markt angeboten werden. Viele gute, wenn nicht die besten und vor allen Dingen aktuellsten Antivirusprogramme sind über öffentliche Netze kostenlos (Freeware) oder gegen eine im Vergleich zum Nutzen geringe Registriergebühr (Shareware) zu beziehen. Die Nutzung einiger Netze und der darauf typischerweise eingesetzten Serversoftware ist ausführlich im Anhang B "Einführung in die Benutzung öffentlicher Netze" beschrieben.

2.5.1 Überwachung (Watcher: "Flushot")

Def. Watcher: Ein Watcher (engl. watcher = Wächter) überwacht das System auf bestimmte Funktionsanforderungen und entscheidet je nach Auftraggeber, Operation, Objekt und Rechten über deren Zulässigkeit.

Funktion. Beim IBM-PC werden alle Betriebssystemfunktionen über sog. Softwareinterrupts aufgerufen; ein Prinzip, das die Basis der Wächterprogramme darstellt und im 3. Kapitel breiten Raum einnimmt. Es sei an dieser Stelle nur soviel dazu gesagt, daß durch das Auslösen eines Interrupts ein bestimmtes Programm, z.B. eine Betriebssystemfunktion, aufgerufen wird, die Parameter übernimmt, die angeforderte Aktion durchführt und das Ergebnis an den Aufrufer zurückgibt. Ein Programm kann Interrupts übernehmen, in dem es den Zeiger, den ein Interruptvektor darstellt, auf eigene Routinen umsetzt. Ein Wächterprogramm kann nun die Aufrufparameter kontrollieren und je nach Beurteilung der Rechtmäßigkeit den Aufruf zulassen und an die Originalfunktion weiterleiten oder aber den Aufruf mit einem simulierten Fehler beenden.

Normalerweise besteht eine Funktionsanforderung aus einem Subjekt, der durchzuführenden Operation und einem Objekt. Das Subjekt ist das aufrufende Programm bzw. der hinter diesem stehende Anwender. Operation und Objekt können beispielsweise das Schreiben einer Datei, das Ändern der Systemzeit oder das Löschen eines Unterverzeichnisses sein. Jedem möglichen Tripel kann das Recht oder das Verbot zur Durchführung zugeordnet werden, was aber wegen der hohen Anzahl der Kombinationen (viele Benutzer, Operationen, Dateien) schnell zu riesigen Rechtedateien führen würde.

Statt dessen ordnet man jedem Teilaspekt des Tripels Rechte zu, deren Kombination dann über die Zulässigkeit der Operation entscheidet. Als Beispiel für diese Methode diene uns das Betriebssystem UNIX, das im Gegensatz zu MS-DOS standardmäßig verschiedene Zugriffsrechte auf Datei- und Verzeichnisebene implementiert.

Beispiel "Dateirechte unter UNIX"
Jede Datei (Objekt) unter UNIX verfügt über jeweils drei Bearbeitungsrechte (Operationen) für drei Anwendergruppen (Subjekte). Bei den Anwendern wird zwischen Eigentümer (**U**ser), Gruppe (**G**roup) und allen anderen (**O**thers) unterschieden. Als

Rechte stehen Lesen (**R**ead), Schreiben (**W**rite) und Ausführen (**E**xecute) zur Verfügung. Ruft ein Benutzer ein Programm auf, so erbt dies seine Rechte und agiert quasi stellvertretend für ihn. Dieses Programm rufe nun eine bestimmte Funktion zur Dateiverarbeitung auf. Das Betriebssystem stellt fest:

1. Das Objekt = den Benutzernamen (hier: `U4475`)
2. Die Operation = die Rechte, die für die Durchführung der angeforderten Operation erforderlich sind (hier: Schreibzugriff)
3. Das Subjekt = die betroffene Datei (hier: `tyrell.corp`)
4. Die Rechte = die mit der Datei verbundenen Zugriffsrechte (hier: `U4511 Judo RWXRWX---`; d.h. Eigentümer `U4511` und Gruppe `Judo` dürfen alles, alle anderen nichts)

Da Aufrufer und Eigentümer des Programms nicht identisch sind und anderen kein Zugriff gewährt wird, müßte der Benutzer `U4475` Mitglied in der Gruppe[8] `Judo` sein, um schreibend zugreifen zu dürfen.

Fallbeispiel. Das Sharewareprogramm `Flushot` überwacht Zugriffe auf Dateigruppen, die Integrität von Dateien und den Zugriff auf Betriebssystemfunktionen. Es gelten einige Einschränkungen bei der Regelung des Dateizugriffs: Es können Dateigruppen (Objekte) und die Art des zulässigen Zugriffs (Operationen) spezifiziert werden, nicht aber Programme (Subjekte), für die diese Rechte gelten oder die eine Ausnahme davon machen dürfen. Das führt dann zu Problemen, wenn Programme kopiert oder von einem Compiler geschrieben werden sollen und der Schreibzugriff auf ausführbare Dateien verboten ist. In diesem Fall gibt es jedesmal einen Fehlalarm, der den Anwender verunsichert und ihn auf die Dauer abstumpft, was bei echter Gefahr zur Mißachtung der Warnungen führen kann.

Schwächen: Direkte Modifikationen. Watcher können nur Funktionen überwachen, in die sie eingeklinkt sind und nur die Parameter bewerten, die sie übermittelt bekommen. Wie wichtig dieses Faktum ist und wie stark es die Fähigkeiten von Wächterprogrammen und die durch sie erreichbare Sicherheit beschränkt, zeigt z.B. das “Hallöchen”-Virus.Dieses verändert vor eigenen Aktionen (z.B. Infektion von Programmdateien) Interruptvektoren direkt im Speicher. Damit wird dieser Vorgang für das Wächterprogramm praktisch unsichtbar, solange es die Vektoren nicht regelmäßig kontrolliert. Doch auch dagegen hat “Hallöchen” etwas in petto: Es verändert für den laufenden Betrieb nicht den Vektor, sondern den Anfang der Routine, auf den dieser zeigt. Dem könnte ein Wächterprogramm wieder mit der Überprüfung seiner Integrität entgegentreten — ein Wettlauf, den jede Seite einmal gewinnt und verliert. Doch darauf allein sollte man sich nicht verlassen.

Schwächen: Direkte Programmierung. Das Prinzip der Umgehung des Betriebssystems ist für jede Funktion anwendbar, die direkt aufgerufen werden kann. So kann ein Virus unmittelbar und ohne den Aufruf von Interrupts Routinen des Betriebssystems anspringen oder den Festplattencontroller programmieren, wenn ihm die

[8] Gruppenzugehörigkeiten werden in der separaten Datei `group` vereinbart.

notwendigen Adressen und Parameter bekannt sind. Die angesprochenen Methoden führen allerdings zu einer mehr oder weniger starken Softwareabhängigkeit von einer bestimmten Betriebssystemversion und zur Hardwareabhängigkeit des Virus vom Rechnertyp. Da PC-Clones auch auf Hardwareebene oft perfekte Kopien des Originals sind, ist es mit dieser natürlichen Begrenzung von Direktaufrufen nicht weit her.

Speicherschutz. Ohne Mechanismen, die Zugriffsschutz auf Speicherebene realisieren, ist solchen Viren, sind sie erst einmal aktiv, nicht beizukommen. Betriebssysteme wie UNIX schirmen die Speicherbereiche gleichzeitig laufender Programme, besonders gegenüber dem Betriebssystems, voneinander ab. Die neueren INTEL-CPUs wie der 80386 und höher bieten ebenfalls Speicherschutz auf Hardwareebene an (Protected Memory). Von dieser Fähigkeit wird unter MS-DOS leider kein Gebrauch gemacht.

Eine besondere Form der Watcher stellen Programme dar, die als ausführbare Datei vorliegenden Programmcode auf verdächtige Funktionsaufrufe und andere suspekte Eigenschaften untersuchen. Dieser Typ wird am Ende des nächsten Abschnitts besprochen, da er eine Kombination der Konzepte "Watcher" und "Scanner" darstellt.

2.5.2 Detektion (Scanner: "F-FChk")

Def. Scanner: Ein Scanner (engl. to scan = etwas absuchen) untersucht Dateien auf bestimmte Codesequenzen und meldet deren Vorhandensein.

Funktion. Die Definition sagt bereits fast alles: Das als Datei vorliegende Programm wird vom Scanner nach bestimmten Merkmalen durchsucht, die von einfachen Bytesequenzen bis zu komplexen Suchmasken reichen. Ein schneller Vergleichsalgorithmus dient dabei vor allen Dingen dem Komfort des Anwenders. Viel wichtiger ist die sorgfältige Auswahl und Anzahl der Suchmuster, die die eigentliche Leistungsfähigkeit eines Scanprogramms bestimmen. Diese Suchstrings müssen so gewählt sein, daß sie einen für das Virus möglichst typischen Programmteil erfassen. Meist wird die Suchinformation kodiert, um den Virenprogrammierern nicht die Gelegenheit zu geben, durch Änderung ihres Machwerks den Scanner ins Leere laufen zu lassen.

Gute Scanner lassen sich mit neuen Viruskennungen nachrüsten, indem der Anwender diese der bestehenden Vergleichsbibliothek hinzufügt. `F-FChk` beispielsweise entnimmt die Vergleichsdaten der ASCII-Datei `sign.txt`, welche die Suchstrings in kodierter Form enthält; für jedes Virus eine Zeile mit Namen und Suchinformation. Neue Viren werden vom Autor untersucht und der Suchstring u.a. über die Diskussionsliste `VIRUS-L` verschickt. Damit vergehen von Entdeckung eines neuen Virus bis zur Verbreitung von Detektionswerkzeugen oft nur wenige Stunden — ein starkes Argument für die Teilnahme an öffentlichen Rechnernetzen.

Schwächen. Die Definition impliziert, daß dem Scanner das Virus, das er zu entdecken in der Lage sein soll, vorher bekannt sein muß. Unbekannte Viren, für die in der Tabelle der Identifizierungsmerkmale kein Eintrag vorliegt, würden nur entdeckt, wenn sie rein zufällig ein schon registriertes Merkmal tragen. Dies ist häufig dann der Fall, wenn das neue Virus eine Weiterentwicklung oder Mutation eines bekannten

Virus ist. Andererseits kann schon eine kleine Veränderung des Virus bewirken, daß der Scanner das Suchmuster nicht mehr findet. Insbesondere sich selbst verändernde Viren der neueren Generation entziehen sich so der Suche. Eine weitere Schwäche ist, daß auch legitime, virenfreie Programme Merkmale enthalten können, auf die der Scanner anspricht. Dadurch kommt es zu Fehlalarmen, die den Überprüfungsaufwand erhöhen und das Vertrauen in das Detektionsprogramm schwächen.

Obwohl die Aufzählung der Kritikpunkte einige Zeilen umfaßte, sind dennoch gute Shareware-Scanprogramme wie `F-FChk` aus Fridrik Skulason's Paket `F-Prot` und `ViruScan` von McAfee Associates in der Lage, bekannte Viren sicher und praktisch ohne Fehlalarme zu identifizieren und, ein wichtiger Punkt, meist auch zu beseitigen. In der Tat dürften Scanner das Gros der Antivirussoftware darstellen. Als Anwender muß man sich aber immer vor Augen halten, daß eine Überprüfung des Dateibestands mit einem Scanner bestenfalls die Abwesenheit von dem Programm *bekannten Viren* garantiert, nicht aber die *Virenfreiheit* des Systems!

Um gute Identifizierungsmerksmale zu erhalten, werden neue Computerviren durch aufwendiges *Reverse Engineering* (Ableitung des Quellprogramms aus dem Programmcode) analysiert. Falls sich dabei zeigt, daß die Veränderungen durch die Infektions- und die Manipulationsfunktion des Virus reversibel sind, kann die Desinfektion von Programmen und Restaurierung von Daten automatisch vorgenommen werden. Bei `F-FChk` wurde diese Funktion integriert, bei `ViruScan` in das Programm `Clean-Up` ausgelagert.

Die oben angeführten Programme überprüfen Dateien nur auf Anfrage durch den Benutzer, der sie für einen Suchlauf starten muß (*Check on Demand*). Das im Paket `F-Prot` enthaltene Programm `F-Driver.sys`, ein Gerätetreiber, untersucht automatisch und für den Benutzer praktisch nicht wahrnehmbar jedes gestartete Programm vor seiner Ausführung auf Viren (*Check on Execute*). Ist es verseucht, wird der Name des Virus angezeigt und der Start abgebrochen. Das Gleiche leisten die Programme `VShield` und `VCopy` von McAfee Associates, die ausführbare Dateien beim Start bzw. beim Kopieren überprüfen. Die Kontrolle beim Kopieren dient der Prophylaxe und verhindert, daß verseuchte Programme überhaupt auf den Rechner gelangen.

Verwandte Konzepte. Ein bereits im Abschnitt "Watcher" angesprochenes, veraltetes Konzept ist die Suche nach verdächtigen Texten und Befehlen in Programmen. Die Idee dabei ist, daß ein z.B. als Textverarbeitung angepriesenes Programm wahrscheinlich keine Texte wie "The Virus-Crew: Got cha, stupid user!!!" und Befehle zur Formatierung der Festplatte enthält. Programme wie das schon betagte `Chk4Bomb` (engl. check for bomb = prüfe auf (logische) Bombe) zeigen jeden längeren Text und verdächtige Funktionsaufrufe des Betriebssystems an, damit der Benutzer sich ein Urteil bilden kann. Dieses Verfahren versucht, einen Virus an seinem Verhalten zu erkennen; Cohen hat sich ebenfalls mit diesem Ansatz beschäftigt, doch dazu später mehr. Dieses Prinzip erlaubt die Entdeckung unbekannter Viren und Trojaner, hat aber zwei gravierende Nachteile. Zum einen muß eine Flut von Hinweisen korrekt interpretiert werden und zum anderen versagt die Suche, wenn die Information kodiert vorliegt, und das ist bei modernen Viren durchweg der Fall.

2.5.3 Schutz der Integrität (Checker: "VTest")

Def. Checker: Ein Checker (engl. to check = kontrollieren) überprüft Dateieigenschaften oder daraus abgeleitete Merkmale anhand einer Vergleichsliste auf Veränderungen.

Funktion. Die dritte Abwehrmöglichkeit besteht im Schutz der Integrität (lat. Unversehrtheit) von Programmen und Daten. Die zugrundeliegende Theorie besagt folgendes:

1. Jedes Virus muß, um ein Programm infizieren zu können, dieses verändern.
2. Jede Veränderung von Dateien ist detektierbar.

In einem sicheren System (engl. trusted system), in dem kein Virus irgendwelche Informationskanäle kontrolliert, wird demnach jeder Virenbefall durch eine detektierbare Veränderung des infizierten Programms gekennzeichnet. Einfach wäre die Duplizierung aller Programme auf ein schreibgeschütztes Medium, um Anhand der Originalprogramme (sog. Spiegeldateien oder Mirror Files) die Arbeitskopien überprüfen zu können. Der dafür benötigte Speicherplatz ist ebenso groß wie für die zu überprüfenden Programme selbst und damit unannehmbar hoch. Es gibt zwei Ansätze, um dieses Problem zu lösen: Entweder man sorgt dafür, daß das Programm nicht verfälscht werden kann oder man versucht, einen möglichst kleinen, aber aussagefähigen "Extrakt" aus dem Programm zu generieren.

Der erste Ansatz geht von einer Verschlüsselung des Programms aus. Ein verschlüsseltes Programm, an dem ein Virus Manipulationen vorgenommen hat, wird bei der Entschlüsselung entweder als Fehlerhaft erkannt oder zerstört. Da weder der Verschlüsselungsalgorithmus noch der verwendete Schlüssel dem Virus bekannt sind, ist eine Infektion fast ausgeschlossen. Weit verbreitete Verschlüsselungsverfahren sind das im diesem Abschnitt noch vorzustellende DES- und das RSA-Verfahren.

Beim zweiten Ansatz hängt alles von der Wahrscheinlichkeit dafür ab, daß zwei verschiedene Dateien dieselbe Prüfsumme besitzen. Quersummen sind denkbar ungeeignet: Die Dateien mit dem Inhalt "42 58" und "69 31" hätten die gleiche Prüfsumme "100"; eine veränderte Datei ließe sich leicht so ergänzen, daß die gleiche Prüfsumme resultiert. Cyclic Redundancy Checks (CRCs), wie sie in der Telekommunikation[9] oft verwendet werden, sind schon erheblich sicherer. Als am sichersten gelten sog. kryptographische Prüfsummen, wie sie z.B. der DES- und der MD4RSA-Algorithmus liefern.

Symmetrische Verfahren. Verschlüsselungsverfahren, bei denen Ver- und Entschlüsselung mit dem gleichen Schlüssel vorgenommen wird, heißen *symmetrische Verfahren*. Das bedeutet, daß beiden Parteien der Schlüssel bekannt sein muß. Dieser Schlüssel ist geheimzuhalten und muß ebenso geheim übermittelt werden, was ein wesentlicher Nachteil des Verfahrens ist. Wenn n Personen miteinander verschlüsselt kommunizieren wollen, müssen $n * (n-1) = n^2 - n$ Schlüssel verteilt werden. Diese Zahl ergibt sich, weil jede der n Personen die Schlüssel der $(n-1)$ anderen kennen muß.

[9] Z.B. im HDLC-Protokoll der ISO-Schicht 2.

Ein bekanntes symmetrisches Verfahren ist der 1974 von IBM entwickelte DES-(**D**ata **E**ncryption **S**tandard)-Algorithmus, der mit einem 64 Bit[10] breiten Schlüssel arbeitet. Das Verfahren gilt als sicher und ist sehr schnell. Hardwareimplementationen wie die der belgischen Firma Cryptech sind in der Lage, 22 Mio Bits pro Sekunde zu verschlüsseln.

Wegen der Probleme mit der sicheren und einfachen Schlüsselverteilung wird DES meistens nur zur Berechnung eines sog. MAC (**M**essage **A**uthentification **C**ode) eingesetzt, einer kryptographischen Prüfsumme. Mit Hilfe des mit einer Nachricht verschlüsselt übertragenen MAC kann festgestellt werden, ob die Nachricht während der Übertragung verändert wurde. Der Empfänger berechnet ebenfalls den MAC und vergleicht ihn mit dem entschlüsselten MAC. Sind beide identisch, wurde die Nachricht nicht manipuliert; die *Integrität* wurde gewahrt. Ein Verfahren für die heute gebräuchliche Verschlüsselung des MAC wird im folgenden Text vorgestellt.

Asymmetrische Verfahren. Dies sind Verschlüsselungsverfahren, die bei Ver- und Entschlüsselung mit unterschiedlichen Schlüsseln arbeiten. Ein Schlüssel, der sog. *Public Key*, darf und kann jedermann bekannt sein und wird z.B. in einer Art Telefonbuch veröffentlicht. Der andere Schlüssel, der *Private Key*, wird wie beim symmetrischen Verfahren geheimgehalten. Dadurch kann entweder genau eine Person die Nachricht verschlüsseln und alle anderen sie lesen oder jeder kann sie verschlüsseln, aber nur einer wieder entschlüsseln.

Die erste Methode eignet sich dazu, sicher den Sender einer Nachricht zu identifizieren. Nur der Sender der Nachricht kann mit seinem Private Key die Nachricht verschlüsselt haben, wenn die Entschlüsselung mit seinem Public Key erfolgreich war. Auf diese Weise läßt sich eine Nachricht *Authentifizieren*; auch die *Integrität* ist gewährleistet. Dieser Modus eignet sich für den Schutz vor Manipulationen, so auch vor Computerviren. Beim Versand von Programmen z.B. über ein Computernetz schlägt dieses Verfahren zwei Fliegen mit einer Klappe: Zum einen ist die Herkunft verifizierbar und zum anderen die Übereinstimmung mit dem Original gewährleistet. Bei der zweiten Methode ist zwar nicht sicher, von wem die Nachricht stammt, aber kein anderer außer dem Inhaber des passenden Private Key kann sie lesen. Auf diese Weise ist die *Vertraulichkeit* gesichert. Eine mögliche Anwendung wäre die Kodierung von Firmen-Telexen, die jede Zweigstelle verschicken, aber nur die Zentrale wieder entschlüsseln kann.

Die bekannteste Methode zur asymmetrischen Verschlüsselung dürfte das von **R**ivest, **S**hamir und **A**dleman entwickelte RSA-Verfahren sein. Es beruht auf der Multiplikation zweier großer Primzahlen, deren Produkt nur mit sehr großem Rechenaufwand wieder in seine Primfaktoren zerlegbar ist. Allerdings wurden in letzter Zeit bedeutende Fortschritte bei der Faktorenzerlegung erzielt, so daß die Sicherheit des RSA-Verfahrens von manchen Experten in Frage gestellt wird [99]. Dennoch erfordert es Superrechner, um einen solchen Code zu brechen; PCs und Viren sind von diesen Leistungen noch sehr weit entfernt.

Elektronische Signatur. Im allgemeinen wird beim RSA-Verfahren ein 512 Bit breiter Schlüssel verwendet, der die Verschlüsselungsgeschwindigkeit allerdings auf $\frac{1}{100}$

[10] Eigentlich nur 56 Bits: 8 Bits sind redundante Prüfinformation.

der Geschwindigkeit von DES herunterbremst. Darum kombiniert man beide Verfahren z.B. bei der *elektronischen Signatur* miteinander [88]. Der schnelle DES-Algorithmus berechnet den nur wenige Bytes umfassenden MAC (ohne Anwendung eines Schlüssels), der mit dem RSA-Verfahren und dem Private Key des Senders verschlüsselt wird. Der Empfänger entschlüsselt mit dem Public Key des Senders den MAC, berechnet ihn seinerseits neu und vergleicht die Ergebnisse miteinander. Sind sie voneinander verschieden, so wurde entweder die Nachricht nicht vom vermeintlichen Sender aufgegeben oder der Inhalt der Nachricht verändert.

Da bei der elektronischen Unterschrift die Nachricht selbst nicht verschlüsselt wird, ist die Vertraulichkeit nicht gewährleistet. Falls dies erwünscht ist, muß die umständliche Prozedur der Schlüsselübermittlung in Kauf genommen werden. Asymmetrische Verfahren können Vertraulichkeit, Integrität und Authentifikation nicht zur gleichen Zeit bieten, falls einer der beiden Schlüssel öffentlich sein soll, was den Vorteil dieser Methode ausmacht.

Anwendung. Cohen schlägt in seinem dritten Artikel "Models of Practical Defenses Against Computer Viruses" [28] ein Betriebssystem vor, das den Start von Programmen nach folgenden Regeln handhabt:

$S3 : \{P, K, S, C : p \times k \Rightarrow S, M, V, k\}$

$$
\begin{aligned}
\text{Programs} \quad P &= \{p_1, p_2, \ldots, p_n\}, n \in \mathbf{I} && (2.1)\\
\text{Keys} \quad K &= \{k_1, k_2, \ldots, k_m\}, m \in \mathbf{I} && (2.2)\\
\text{Checksums} \quad S &= \{s_1, s_2, \ldots, s_o\}, o \in \mathbf{I} && (2.3)\\
\text{Transformation} \quad C & \quad C : P \times K \Rightarrow > S && (2.4)\\
\text{Moves} \quad M &= \{m_1, m_2, m_3, m_4\} && (2.5)\\
\text{Values} \quad V &= \{v_1, \ldots, v_n\}\\
& \quad \forall v_i \in V,\\
& \quad \exists s_i \in S : v_i = s_i && (2.6)\\
\text{secret key} & \quad k \in K && (2.7)\\
\text{Times} \quad T &= \{t_1, t_2, \ldots, t_n\}, n \in \mathbf{I}\\
V &= \{v_1, \ldots, v_n\}\\
& \quad \forall p_i \in P \quad \text{wobei}\\
& \quad \forall v_i \in V,\\
& \quad v_i = C(p_i, k) \quad \text{zum Zeitpunkt} \quad t_i && (2.8)\\
& \quad t_j : \quad \forall t_i \in T, \quad t_j > t_i && (2.9)
\end{aligned}
$$

Das sieht beeindruckend aus — aber was steckt dahinter? Fangen wir von oben mit der Definition der Mengen und Operationen an. Es existiert eine Menge P mit n Programmen, die den kompletten Programmbestand eines Rechnersystems darstellt. Für jeden der m Benutzer existiert ein geheimer Schlüssel k_i, die zusammen die Menge aller verwendeten Schlüssel K bilden. Die Funktion C übernimmt Programm und

Schlüssel als Parameter und liefert die Prüfsumme zurück, die Element der Menge aller möglichen Prüfsummen S ist. V ist ein Satz der Prüfsummen aller Programme zu einem bestimmten Zeitpunkt.

Nach den mathematische Präliminarien kommt der eigentlich interessante Teil. Jedem Zeitpunkt t_i aus der Menge aller Zeitpunkte T ist ein Satz V von Prüfsummen zugeordnet, der den aktuellen Inhalt aller Programme widerspiegelt. Zur Bildung von V wird für jedes Programm p_i die Prüfsumme v_i berechnet, indem die Funktion C auf dieses Programm und den Schlüssel des Benutzers angewendet wird. t_j ist einfach ein Zeitpunkt, der später als t_i liegt.

Das Betriebssystem S3 arbeitet wie folgt:

1. Zum Zeitpunkt t_j, der also nach der Prüfsummenberechnung liegt, wird der Start des Programms p_i angefordert
2. Falls $C(p_i, k) = v_i$ ist (die aktuelle Prüfsumme ist gleich der gespeicherten; p_i ist unverändert), dann führe p_i aus und gehe zu 1
3. Das Programm wurde verändert. Frage den Benutzer nach der Verfahrensweise m:

   - $m = m_1$: Gehe zu 1
     → führe Programm nicht aus (sicher)
   - $m = m_2$: Führe p_i aus; gehe zu 1
     → führe Programm trotzdem aus (unsicher)
   - $m = m_3$: Setze $v_i = C(p_i, k)$; führe p_i aus; gehe zu 1
     → berichtige Prüfsumme, führe Programm trotzdem aus (unsicher)
   - $m = m_4$: Lade Zustand von p_i zum Zeitpunkt der Prüfsummenberechnung; führe p_i aus; gehe zu 1
     → lade unveränderte Sicherungskopie zurück, führe Programm aus (sicher)

Der Vorläufer von S3, das Betriebssystem S2, erkennt Veränderungen nicht an einer Prüfsumme, sondern an unterschiedlicher gespeicherter und aktueller letzter Bearbeitungszeit einer Datei. Für die Anwendung dieser Strategie muß natürlich sichergestellt sein, daß jede Schreiboperation auch gleichzeitig die Zeitmarke der letzten Bearbeitung verändert. Ein Virus könnte dies umgehen, niemals jedoch die Veränderung des Dateiinhalts. Deshalb eignet sich S2 nur für vertrauenswürdige Systeme, “Trusted Systems”. Das vorgestellte Betriebssystem S3 eignet sich auch für nicht vertrauenswürdige Rechner, die “Untrusted Systems”. Auf die Ausnahme, die Stealth-Viren darstellen, kommen wir weiter unten bei der Diskussion der Schwächen noch zurück.

Cohen hat die Prüfsummenfunktion noch so erweitert, daß auch die Abhängigkeiten der Programme (S4) und Daten (S5) untereinander berücksichtigt werden. Diese Relationen werden entweder eingegeben oder im laufenden Betrieb durch ausprobieren ermittelt. Falls das Programm p_i gestartet wird, überprüft das Betriebssystem rekursiv auch alle Programme und Daten, von denen es abhängig ist. Damit wird die Integrität der bearbeiteten Information, des verarbeitenden Programmes und damit seiner Ausgaben sichergestellt. Andererseits wird die Menge der Relationen schnell sehr groß,

komplex, undurchschaubar und unwartbar. Bei jedem Programmstart muß u.U. eine Vielzahl von Dateien überprüft werden, was die Leistung des Systems in Bezug auf andere Aufgaben vermindert.

Besonders elegant ist der Selbsttest beim Start eines Programms. Dies funktioniert nur bei Programmen, die sich nicht selbst verändern wie z.B. Turbo-Pascal vor Borland oder `system.exe` auf VMS-Systemen. Viele Antivirusprogramme wenden diese Methode an und überprüfen beim Start, ob sie nicht selbst Opfer einer Infektion oder sonstigen Manipulation geworden sind. Das Verfahren bietet zwar keinen Schutz vor Verseuchung, warnt aber den Benutzer vor weiteren Folgen. `CAware` (von engl. "C" aware = aufmerksames "C") ist eine als Objektdatei vorliegende "C"-Funktion, die in jedes Programm eingebunden werden kann und einen Selbsttest wie oben beschrieben realisiert. Ein separates Programm berechnet die endgültige Prüfsumme und trägt sie in das mit `CAware` ausgerüstete Programm ein.

Schwächen. Die Detektion von Veränderungen ist auf einem sauberen System sicher möglich. Bevor ein Check-Programm eine Manipulation feststellen kann, muß diese erst einmal eingetreten sein. Das bedeutet, daß die Anwesenheit eines Virus im System frühestens nach einer erfolgten Infektion oder Manipulation nachweisbar ist. Liefert der Checker keinen Hinweis auf eine Veränderung, läßt dies zwei Schlüsse zu: Entweder ist das System virenfrei oder das System ist verseucht und das Virus verhält sich ruhig.

In manchen Fällen führt ein nachträglich in ein Programm eingebauter Selbsttest zu Schwierigkeiten. Manche Programme besitzen die Unart, sich selbst zu verändern oder werden durch Konfigurationsprogramme manipuliert. Ein Selbsttest bringt daher keinen Informationsgewinn in Sachen Sicherheit. Andere Programme überprüfen bereits "ab Werk" ihre Integrität. Diese geht durch das Eintragen der Prüfsumme für den Selbsttest verloren, das Programm verweigert die Arbeit.

Ganz anders ist die Situation in einem System, auf dem ein Virus bereits aktiv ist. Wie bei den Watchern (s. dort) kann ein Checker nur das überprüfen, was ihm die Betriebssystemfunktionen liefern. Stealth-Viren besitzen die Eigenschaft, den Zugriff auf eine verseuchte Datei abzufangen, diese temporär zu desinfizieren und dann erst weitere Operationen zuzulassen. Dadurch erscheint das Programm bei einer Überprüfung und *aktivem* Virus unverseucht — eine geradezu paradoxe Situation. Wichtig ist deshalb bei allen Maßnahmen gegen Viren, daß sich zum Zeitpunkt der Systemüberprüfung kein Virus aktiv im Speicher befindet, das System also "sauber" ist.

Aus eigener Erfahrung. Zum guten (?) Schluß noch eine reale Begebenheit, die verdeutlichen soll, daß alle Schutzmaßnahmen nichts nutzen, wenn der Anwender Warnungen nicht korrekt zu deuten weiß. Ende 1989 sicherte ein Studienkollege des Autors seinen Rechner mit einem Watcher (`Flushot`), einem Checker (`VTest`) und einem Scanner (`ViruScan`). Alle drei Programme arbeiteten, wie sich nachträglich herausstellte, einwandfrei. Der Checker meldete keine Veränderungen, der Scanner fand nichts, nur der Watcher monierte gelegentlich, daß ein gerade gestartetes Programm versuchte, auf ein anderes Programm zuzugreifen. Das hätte natürlich jeden stutzig machen müssen, aber gerade der Watcher gab so oft falschen Alarm, daß auch diese Warnung ignoriert wurde.

Die rein zufällige Untersuchung eines Programms mit `debug` zeigte das Ausmaß der Fehleinschätzung: Der Rechner war vollständig mit dem "Hallöchen"-Virus verseucht, so daß in Ermangelung potentieller Opfer gar keine Verbreitung mehr möglich war → der Checker schwieg. Dazu kam, daß dieses spezielle Virus eine deutsche Erfindung ist und von der damaligen Version von `ViruScan` noch nicht erfaßt wurde → auch der Scanner blieb deshalb ruhig. Der Autor nahm daraufhin Kontakt mit Fridrik Skulason auf, dessen nächste Version seines Programmpakets `F-Prot` das Virus entdecken und beseitigen konnte.

2.6 Analogien zur Biologie

Der Begriff "Virus" wurde der Biologie entlehnt, weil die Analogie das Verhalten eines Computervirus in Funktion und Ausbreitung gut beschreibt. Es stellt sich die Frage, ob Erkenntnisse der Biologie bei der Virenbekämpfung auch für die Eindämmung von Computerviren verwendet werden können. In sehr direkter Form fand dieser Gedanke bereits im Aufsatz "The IPM Model of Computer Virus Management" von S.Jones und E.White Anwendung [31]. Die Autoren schlagen vor, das aus der Agrarwissenschaft stammende **I**ntegrated **P**est **M**anagement-Modell (IPM) auf Computer zu übertragen. Der Gedanke: Bereits existierende Erkenntnisse, wirksame Analysemethoden und Verfahren aus der Landwirtschaft nutzen, um dadurch evtl. neue Möglichkeiten zur Virenabwehr zu finden.

Ein Vergleich zwischen biologischen und programmierten Viren deckt interessante Parallelen auf, die in diesem Abschnitt untersucht werden [85]. Allerdings fängt die Gegenüberstellung mit einem Unterschied an. Während biologische Viren im Laufe der Evolution durch Mutation und Selektion entstanden sind, werden Computerviren erst seit kurzer Zeit von Menschen programmiert und haben nicht die Fähigkeit, sich selbst zu vervollkommnen. Das Aufkommen solcher Viren stellt aber eine mögliche zukünftige Entwicklung dar.

Transport (Vektor). Viren benutzen einen lebenden Wirt, ohne den sie sich nicht verbreiten und Aktivitäten entfalten können. Auf einen Computer übertragen bedeutet diese Eigenschaft, daß ein Virus in einem Objekt befinden muß, das auf irgendeine Art und Weise zur Ausführung gebracht werden kann. Demnach hat das Virus zwei prinzipielle Möglichkeiten, einen MS-DOS-Rechner zu befallen:

1. Durch Infektion von Programmdateien oder Daten, die durch Programme interpretiert werden (fertige Kompilate, Zwischenprodukte wie Objektdateien und Bibliotheken, Quelltexte).
2. Durch Infektion von besonderen Programmen (Bootsektor einer Diskette oder Festplatte).

Fast alle modernen Viren verwenden die erste Methode, weil alle Programmdateien als Infektionsträger nutzbar sind und deshalb die Vermehrung auf breiter Basis erfolgen kann. Dateien werden auf Speichermedien wie Disketten (magnetisch, optisch)

oder per DFÜ (Datenfernübertragung) transportiert. Durch den gerade auf dem PC-Sektor weit verbreiteten Austausch von Programmen (bes. Raubkopien, Share- und Freeware) werden Infektionen schnell verbreitet.

Bei der zweiten Methode ist das Virus auf den Austausch von Disketten oder Wechselplatten *und* das Urladen von diesen angewiesen. Da das vergleichsweise selten geschieht, wenn sich eine Festplatte im System befindet, verbreiten sich solche Viren nur in geringen Raten. Dennoch waren Urlader-Viren die ersten Computerviren überhaupt, weil das Prinzip einfach zu realisieren ist und in der Anfangszeit der PCs Festplatten nicht standardmäßig zum System gehörten. Trotz der angeführten Hemmnisse ist zu beobachten, daß auch in neuerer Zeit moderne Bootviren mit Stealth-Eigenschaften immer wieder Rechner verseuchen. Hauptgrund dafür dürften Disketten sein, die beim Ausschalten des Rechners irrtümlich im Laufwerk vergessen wurden. Beim Einschalten wird dann das Betriebssystem unfreiwillig von Diskette geladen, falls der Anwender die Diskette nicht vorher entdeckt.

Inkubationszeit. Die Zeit zwischen Infektion und dem Auftreten der ersten Symptome wird als Inkubationszeit bezeichnet. Eine lange Verzögerung zwischen Infektion und Ausbruch der Krankheit bewirkt, daß u.U. viele Individuen Opfer einer Ansteckung werden, bevor dies durch Auftreten von Symptomen bemerkt wird. Ein Computervirus mit einer langen Inkubationszeit und geringer Infektiosität, das sich insgesamt sehr unauffällig verhält ("schleichende Infektion"), hat gute Ausbreitungschancen, falls Watcher und Scanner als primäre Mittel zur Abwehr eingesetzt werden. Checker können auch diesen Typ sofort erkennen, sobald das Virus eine Datei verändert hat.

Auch der umgekehrte Weg kann schnellen Erfolg bringen, wie die Geschichte beweist. Die im Mittelalter wütende Pest führte innerhalb von wenigen Stunden nach der Infektion zum Tode. Die Seuche breitete sich wegen der großen Anzahl von Vektoren (Flöhe auf Ratten), mangelhafter Hygiene und genereller Unkenntnis der Ursachen rasch aus. Als fast perfekte Analogie hierzu wäre der "INTERNET-Worm" zu nennen: Schnelle Überlastung der befallenen Rechner, rasche Verbreitung über eine Vielzahl von Netzwerkverbindungen, mangelhafte Systemsicherheit und schließlich die Unkenntnis mancher Systemadministratoren führten zum Zusammenbruch vieler Systeme.

Infektiosität. Die theoretisch mögliche Ausbreitungsgeschwindigkeit einer Infektion ist durch die Infektiosität bestimmt. Biologische Viren sind während ihrer Vermehrung in der Wirtszelle praktisch nicht vorhanden. Das infizierende Virus hat seine DNA abgegeben und ist nur noch eine leere Proteinhülle, die Nachkommen befinden sich noch im Bau. Die Ansteckung kann nicht weitergegeben werden, bevor die manipulierte Zelle platzt und die neuen Viren freigibt. Computerviren dagegen sind sofort beim ersten Start zur Infektion anderer Programme fähig. Andererseits kann es der Programmierer aus Tarnungsgründen für wünschenswert erachtet haben, daß erst nach Ablauf einer gewissen Zeitspanne oder Anzahl von Ereignissen eine Verbreitung erfolgt.

Schaden. Ein Virus schädigt einen Organismus auf drei Arten:

Biologische Viren	Computerviren
1. Die vom Virus zur Verbreitung benutzten Zellen werden in ihrer Funktion zunächst beeinträchtigt und schließlich bei der Freisetzung der Viren zerstört.	1. Programme werden bei der Infektion verändert, wenn nicht sogar zerstört (z.B. durch überschreibende Viren).
2. Manche Bakterien (z.B. Botulinus) produzieren als Nebeneffekt der Infektion Gifte, die den Organismus schädigen.	2. Manche Viren tragen Schadensfunktionen mit sich, die Soft- und Hardware schädigen sowie Dienste blockieren können.
3. Durch die Abwehrmaßnahmen wird der Organismus belastet.	3. Alle Antivirusprogramme verbrauchen ihrerseits Rechenzeit und Speicherplatz.

Reservoir. Als Reservoir einer Infektion bezeichnet man alle Orte und Gegenstände, in oder auf denen sich der Erreger halten kann. Viren sind durch ihre einfache Bauweise auch ungünstigsten Bedingungen gegenüber resistent und können in kristalliner Form Vakuum, große Hitze und Kälte außerhalb eines Wirtes überleben. Analog dazu ist jede Diskette, die ein verseuchtes Programm enthält und in irgendeiner Schublade vergessen ihr Dasein fristet, ein Reservoir für eine erneute Infektion.

Die Gefahr der Reinfektion ist ein wesentliches Problem bei der Desinfizierung von Computern, denn jedes einzelne z.B. auf Sicherheitskopien übersehene Virus kann eine erneute Verseuchung auslösen. Ein spezielles Problem ist der Warmstart eines Rechners, den ein Virus möglicherweise unbeschadet im Hauptspeicher überstehen kann. Um dies auszuschließen, sollte ein Kaltstart durchgeführt werden, bei dem der Rechner mind. eine Minute ausgeschaltet wird, damit die flüchtigen Speicherbausteine ihren Inhalt sicher verlieren. Durch Laden des Betriebssystems von einer sauberen Diskette ist die Virenfreiheit des Rechners garantiert. Vor dem Einlesen von Backups sollten diese auf Viren überprüft werden.

Immunität. Beschäftigen wir uns nun mit der Abwehr von Krankheitserregern. Immunsysteme lebender Organismen wehren Viren ab, indem sie spezifische Antikörper bilden, die die Eindringlinge bekämpfen. Durch Impfungen wird der Organismus angeregt, bereits Antikörper gegen Erreger zu bilden, die ihm noch gar nicht begegnet sind. Die meisten Antivirenprogramme (Scanner) besitzen eine Art starres Abwehrsystem, das Viren an besonderen Merkmalen erkennt. Es macht ihre Anwesenheit dem Benutzer bekannt oder vernichtet sie. Während ein biologischer Organismus aber von selbst dazulernt und ihm unbekannte Viren (auch Mutationen) bekämpfen kann, versagt hier gemäß Cohen's Theorien das programmierte Immunsystem. Die Erweiterung des Wissens kann bei manchen Programmen wie z.B. `F-Prot` und `ViruScan` manuell durch den Benutzer erfolgen, indem dieser der Tabelle der Erkennungsmerkmale weitere Einträge hinzufügt.

Cohen versteht unter der Immunisierung eines Programms, daß die Programmdatei mit dem "ist infiziert"-Merkmal des abzuwehrenden Virus versehen wird. Dieses Verfahren versagt aus zwei Gründen: Einmal widersprechen sich z.T. die Kennungen

verschiedener Viren und zum anderen ignorieren manche Viren (evtl. nach einer bestimmten Strategie) eine bereits erfolgte Infektion.

Es ist unmöglich, einen perfekten universellen Virusdetektor (UVD) zu programmieren, der über ein Programm eine Aussage machen kann, ob es einen Virus enthält oder nicht. Wie leider das AIDS auslösende Retrovirus HTLV beweist, sind auch biologische Immunsysteme nicht unfehlbar, wenn die Abwehr selbst zum Opfer wird. Analog dazu sind viele der moderneren Computerviren in der Lage, Schutzprogramme zu umgehen, wenn sie erst einmal aktiviert worden sind. Schlimmer noch: Da Scanner auf jedes Programm zu Prüfzwecken zugreifen, bekommt ein aktives "Infect on Open"-Virus jede infizierbare Datei quasi auf dem Präsentierteller angeboten.

Isolation. Ein andere Abwehrmaßnahme ist das Anlegen eines Schutzanzuges, der vor jeglicher Einwirkung von außen schützt und z.B. Menschen ohne intaktes Immunsystem das Überleben ermöglicht. Aufwendige Filter sorgen dafür, daß die Atemluft keine gefährlichen Krankheitserreger oder Allergene enthält. Für Rechner heißt das: Verseuchte Programme werden gar nicht erst in das System gelassen, sondern abgewiesen, bevor Schaden möglich ist. Gerade für MS-DOS-Rechner, die über kein "Immunsystem" (Zugriffskontrolle) verfügen, scheint dieses Schutzkonzept ein vielversprechender Weg zu sein. Generell gilt, daß es immer einfacher ist, prophylaktisch tätig zu werden als erst Maßnahmen zu ergreifen, wenn schon etwas passiert ist.

Benutzer privater Computer werden eigenverantwortlich darauf achten, daß sie keine infizierten Programme einschleppen. Wie sieht die Sachlage bei Rechenzentren und betrieblich genutzten Rechnern aus? Wenn man davon ausgehen könnte, daß ausschließlich die virenfreien Dienstprogramme des Rechenzentrums (Compiler, Editoren etc.) benutzt werden und die Benutzer nur Quelltext kompilieren, der selbst natürlich kein Virus enthalten darf, wäre eine Verseuchung nicht möglich. Die Rechner sind dann auf kontrollierte Art und Weise von der Außenwelt isoliert, ohne unbenutzbar zu werden.

Die Anbieterseite. Für die Virenfreiheit der Dienstprogramme kann das Rechenzentrumspersonal garantieren, wenn gewisse Regeln beachtet werden. Fälle, in denen Originalsoftware verseucht war, sind bisher auf Disketten, die z.B. Büchern beigefügt waren, begrenzt geblieben ([38] 3.309, 315, 322, 324: "The Fractal System"). Die Rechenzentrumsseite stellt demnach keine Verseuchungsquelle dar, zumal sie selbst daran interessiert ist, die Rechner virenfrei zu halten.

Die Benutzerseite. Für die Benutzer des Rechenzentrums kann eine Benutzerordnung festgelegt werden, die es erlaubt, Quelltext einzuspielen, mit den auf dem System vorhandenen Programmen zu bearbeiten und zu speichern. Nicht erlaubt hingegen ist es, das Betriebssystem von Diskette zu laden sowie ausführbare Dateien mitzubringen und ablaufen zu lassen. Die Einhaltung der Verhaltensregeln läßt sich jedoch aus verschiedenen Gründen nicht ständig kontrollieren. Die Verletzung der Vorschriften geschieht, wie in 2.1.3 "Motive der Anwender" untersucht, unbewußt oder absichtlich. Der nächste Abschnitt stellt Konzepte vor, mit deren Hilfe kontrollierte Isolation und damit der Schutz vor sicherheitsgefährdender Software realisiert werden kann.

2.7 Alternative Konzepte

Ein vielversprechender Ansatz analog zu Verfahren aus der Biologie ist der eines "Schutzanzuges" für den Rechner, d.h. die *kontrollierte Isolation* des Systems von Programmen von außerhalb. Dieses Schutzprinzip ist vor allen Dingen für APCs in einem Betrieb oder Rechenzentrum gedacht, die vor Computerviren und auch gegen Verseuchungsversuche der Anwender geschützt werden sollen. Für den PC zu Hause kann angenommen werden, daß der Benutzer ein eigenes Interesse an der Virenfreiheit seines Rechners hat und deshalb nicht gegen die Schutzprogramme arbeitet. Kontrollierte Isolation stellt im diesem Moment eine Selbstbeschränkung, eine Einschränkung des freien Arbeitens dar. Sie kann aber zu erhöhter Sicherheit beitragen, indem der Benutzer vor sicherheitsgefährdenden Operationen gewarnt wird und dann selbst entscheiden kann, ob er fortfahren möchte oder nicht.

2.7.1 Schutzzonen und Kontrollpunkte

Abbildung 2.2 stellt schematisch den Aufbau eines Computersystems unter dem Gesichtspunkt "Transport von Dateien" dar. Die CPU ermöglicht die Ausführung eines Programms, das eine Datei innerhalb des Dateisystems ist. Dieses besteht aus statischen Teilen wie der Festplatte (intern) und auswechselbaren Datenträgern wie Disketten und Wechselplatten (extern), die nur temporär zum Dateisystem gehören. Der Bestand und Zustand von Dateien auf Festplatte verändert sich durch

- Kopieren von Dateien von externen auf interne Datenträger,
- Eingabe von Daten und Quelltexten per Tastatur,
- Datenübertragung per DFÜ und
- interne Manipulationen (umbenennen, verändern, kompilieren von Dateien).

Das System kann in Schutzzonen unterschiedlicher Restriktivität unterteilt werden, die verschieden weit gefaßte Sicherheitsbereiche um den Prozessor bilden. Je höher die Nummer, desto umfassender der Schutz und eingeschränkter die Manipulationsmöglichkeiten. Der Prozessor (Schutzzone SZ I) ist der Ort, an dem ein passives, gespeichertes und harmloses Virus durch die Ausführung eines Programms zu einem aktiven wird, das Daten und Hardware bedroht. Auszuführende Programme, die potentielle Infektionsträger sind, werden vor dem Ablauf kontrolliert. SZ II beinhaltet Programme auf Festplatte, die als sauber gelten. Zu diesem Bestand dürfen keine Programme hinzukommen, denn dadurch könnten passive Viren ins System gelangen. Unterstützt wird diese Forderung durch SZ III, die Programmdateien vor Manipulationen schützt. Dazu zählt auch die Überwachung von (dem Namen nach) nicht ausführbaren Dateien. Diese könnten immerhin ein "getarntes" Virus enthalten und durch Umbenennung ausführbar werden.

In den folgenden Abschnitten wird untersucht, welcher Ansatz diesen Schutzzonen zugrunde liegt, wie sie realisiert werden können und wo Stärken und Schwächen der Konzepte liegen.

Abbildung 2.2: Schutzzonen und Kontrollpunkte

2.7.2 Generelle Verbote

Kontrollpunkt “0” (s.Abb. 2.2): Start von Programmen auf Diskette

Ansatz: Dieser Kontrollpunkt der Schutzzone I bildet die erste “Verteidigungslinie” und verhindert, daß Programme von außerhalb direkt ausgeführt werden. Diese kommen auf austauschbaren Datenträgern (Disketten, Wechselplatten) ins System. Grundsätzlich ist der Start von Programmen von Diskette unzulässig, da bei diesen nicht sicher ist, ob sie keine Viren enthalten oder mißbräuchlich genutzt werden. Es muß ebenfalls unmöglich sein, das Betriebssystem von Diskette zu laden. Das gilt besonders unter dem Aspekt des Selbstschutzes des Wächterprogramms.

Bei der Programmeingabe über DFÜ oder Tastatur ist bis zur Ausführung ein Zwischenschritt erforderlich. Bei der Datenfernübertragung wird das Programm zunächst in einer Datei abgelegt und kann nicht direkt “von der Leitung” gestartet werden. Diese Methode der Einschleusung wird von Schutzzone II abgedeckt. Um von der Eingabe von Quelltext per Tastatur zu einem lauffähigen Programm zu gelangen, ist der Einsatz eines Compilers oder Interpreters erforderlich. Wie schon erwähnt, kann gegen die Eingabe von Tastatur nichts unternommen werden, es sei denn, daß für den bestimmungsgemäßen Betrieb keine Software zur Programmerstellung notwendig ist und deshalb nicht installiert wird.

Maßnahmen: Betriebssystemaufrufe zum Programmstart werden abgefangen und der Programmname bezüglich des Laufwerks überprüft. Startversuche von Diskette werden abgebrochen. Dieses Vorgehen impliziert die Verwendung eines Wächterprogramms (Watchers), das sich in die relevanten Funktionen einklinkt.

2.7.3 Strikte Isolation

Kontrollpunkte "2":

1 Kopieren von Programmen

2 Kopieren von Daten als Programm. Evtl. Enttarnvorgang

3 Kopieren von Daten als Programm oder umbenennen, festplattenintern

Ansatz: Alle Programme, die sich auf Festplatte befinden, sind validiert; es dürfen deshalb keine Programme von außen (Diskette, Datenfernübertragung, Tastatur) hinzukommen.

Der Begriff *Validierung* kommt vom angelsächsischen to validate = für zulässig, gültig erklären. Ein *validiertes* Programm ist von einer übergeordneten Instanz, dem Anwender oder stellvertretend dem Sicherheitssystem, für die Ausführung zugelassenen worden. SZ II überwacht und isoliert den Programmbestand auf Festplatte. Durch diese Maßnahme werden Operationen verhindert, die den Rechner bereits infizieren könnten, was das Einspielen von passiven Viren (d.h. verseuchten Programmen) angeht. Falls keine explizite Validierung durchgeführt wird (s. SZ I), würde der Start eines verseuchten Programms das Virus aktivieren.

Maßnahmen: Alle Kommandos und Funktionen, die ausführbare Dateien transportieren, schreiben oder umbenennen können, sind zu überwachen. Jeglicher Schreibzugriff auf ausführbare Dateien ist zu verweigern.

2.7.4 Kontrollierte Isolation

Kontrollpunkte "2":

1 Kopieren von Programmen

2 Kopieren von Daten als Programm. Evtl. Enttarnvorgang

3 Kopieren von Daten als Programm oder umbenennen, festplattenintern

Ansatz: Alle Programme, die sich auf Festplatte befinden, sind validiert; es dürfen deshalb keine Programme von außen (Diskette, Datenfernübertragung, Tastatur) hinzukommen. Nur Programme mit Schreibberechtigung dürfen ausführbare Dateien schreiben.

Theoretisch könnte man, wie angeführt, generell verbieten, daß ausführbare Dateien geschrieben oder verändert werden können. Allerdings ergeben sich hierbei zwangsweise Lücken, wenn der Rechenzentrumsbetrieb nicht lahmgelegt werden soll: Compiler, Linker und ähnliche Programme müssen in der Lage sein, Programme zu schreiben. Auch festplatteninternes Kopieren ausführbarer Dateien sollte möglich sein. Dies gilt nicht für Arbeitsplätze, an denen keine Programmerstellung erfolgt, sondern Software lediglich angewendet wird. In diesem Fall ist die bereits angesprochene strikte Isolierung möglich.

Einen Ausweg bietet die Kontrolle der Schreibzugriffe auf ausführbare Dateien an. Bei jedem Zugriff wird geprüft, ob das Programm dazu berechtigt ist (z.B. Compiler des Rechenzentrums bei der Kompilierung auf Festplatte). Ein Hauptnachteil dabei ist, daß eine Liste der dazu validierten Programme angelegt, gewartet und geschützt werden muß. Dazu kommt die Schwierigkeit, unter MS-DOS von der Anforderung eines Schreibzugriffs auf das aufrufende Programm zu schließen. Mit dieser Technik werden wir uns noch ausgiebig in den folgenden Kapiteln beschäftigen.

Maßnahmen: Alle Kommandos und Funktionen, die ausführbare Dateien transportieren, schreiben oder umbenennen können, sind zu überwachen. Je nach anforderndem Programm, Ort der Herkunft, Zielort und Dateityp ist die Operation zu verweigern oder zuzulassen.

2.7.5 Offenes System (Explizite Validierung)

Kontrollpunkt "1": Start von Programmen auf Festplatte

Ansatz: Nur Programme, die als validiert gekennzeichnet sind, können gestartet werden; Programme und Daten sind beliebig kopierbar.

Schutzzone I schützt den Rechner vor dem Start verseuchter Programme. Wenn ein Programm ausgeführt werden soll, wird zunächst anhand einer Tabelle die Berechtigung dazu geprüft, die vom Systemadministrator vergeben wird. Ansonsten ist es möglich, beliebige Daten und Programme auf Festplatte zu kopieren (daher "offen"); die Menge und Art der Dateien auf Festplatte ist veränderlich.

Einen Nachteil dieser Methode stellt die Tabelle der validierten Programme dar, die zu erstellen, modifizieren und vor unberechtigtem Zugriff zu schützen ist. Weitere Probleme entstehen dadurch, daß die bloße Namensänderung eines Programms aus einem nicht validierten ein zulässiges Programm machen kann.

Maßnahmen: Betriebssystemaufrufe zum Programmstart werden abgefangen und der Programmname bezüglich des Berechtigung überprüft. Unzulässige Startversuche werden abgebrochen.

2.7.6 Offenes System (Wahrung der Integrität)

Kontrollpunkt "1": Start von Programmen auf Festplatte

Ansatz: Kodiert gespeicherte Programme werden beim Start dekodiert; Programme und Daten sind beliebig kopierbar.

Diese alternative Form der SZ I schützt den Rechner vor der Ausführung manipulierter Programme. Jedes Programm wird bei seiner Installation durch Rechenzentrumspersonal verschlüsselt. Bei geeigneter Wahl des Kodierungsverfahrens ist es praktisch unmöglich, ein Programm erfolgreich zu dekodieren. Dadurch kann das Programm weder modifiziert (Wahrung der Integrität) noch "für den Hausgebrauch" kopiert werden (Kopierschutz). Die Überwachungsfunktion dekodiert das aufgerufene Programm, bevor es die Kontrolle und das Originalprogramm an das Betriebssystem übergibt. Diese Methode ist insbesondere bei Betriebssystemen ohne Zugriffskontrollen wie MS-DOS sinnvoll und erfolgreich einzusetzen und kann theoretisch auf beliebige Dateien ausgeweitet werden. Es stehen heute bereits Hardwarebausteine zur Verfügung, die die Verschlüsselung/Entschlüsselung von großen Datenmengen in kurzer Zeit bewerkstelligen können, ohne dabei die CPU nennenswert zu belasten.

Alternativ dazu kann das Programm mit einer Signatur versehen werden, die bei der Installation auf Festplatte von einem Kodieralgorithmus über das Programm erzeugt und beim Start überprüft wird (s.a. "Checker"). Damit fällt allerdings die Möglichkeit eines Kopierschutzes weg. In beiden Fällen darf das Programm vor der Kodierung mit keinem Virus behaftet sein, weil sonst der Viruscode als legitimer Bestandteil des Programmes mitgesichert wird. Dennoch würden alle weiteren Infektionen sicher erkannt.

Maßnahmen: Betriebssystemaufrufe zum Programmstart werden abgefangen und die Programmdatei dekodiert bzw. die berechnete Signatur mit dem Sollwert verglichen. Unzulässige Startversuche werden abgebrochen.

2.7.7 Zusätzliche Maßnahmen

Kontrollpunkte "3":

1 Kopieren von Daten. Es wird per Dateianalyse überprüft, ob als "Daten" identifizierte Dateien tatsächlich nur Text enthalten.

2 Kopieren von Programmen als Daten (Tarnvorgang)

3 Kopieren von Programmen als Daten oder Umbenennen, festplattenintern

Ansatz: Alle Operationen, die für den bestimmungsgemäßen Betrieb nicht erforderlich sind, werden vorbeugend nicht zugelassen.

Schutzfunktionen der SZ III betreffen den kompletten Datenbestand der Festplatte. Die hierdurch abgefangenen Operationen betreffen zwar nicht direkt Programme, können aber Vorschub zu illegalen Manipulationen leisten (Kopieren von getarnten Programmen, "tarnen" und "enttarnen").

Maßnahmen: Analog zu den unter Schutzzone II angesprochenen Maßnahmen müssen zusätzlich zu kopierende Daten auf ihren Inhalt überprüft werden. Operationen, bei denen Abweichungen zwischen vorgeblichen und tatsächlichem Dateityp vorliegen oder hervorgerufen würden, sind zu verweigern.

2.7.8 Vergleich der Konzepte

In diesem Abschnitt werden konventionelle Sicherheitssysteme, Antivirusprogramme und die alternativen Konzepte gegenübergestellt und miteinander verglichen.

Konventionelle Sicherheitssysteme. Zugriffskontrollsysteme gebräuchlicher Art greifen genau dann nicht, wenn Viren nur das tun, was sie bzw. der Benutzer, dessen Rechte sie sich bedienen, auch tun darf. Wie Cohen gezeigt hat, sind solche Viren ohne weiteres in der Lage, die Schutzeinrichtungen von Mini- und Mainframebetriebssystemen in Minuten oder höchstens Stunden zu unterlaufen. Was bleibt dann von der gegebenen oder angeblichen Sicherheitsüberlegenheit der Mainframes und Minis gegenüber den APCs übrig?

Zumindest werden Logdateien geführt, die protokollieren, wer wann welche Operation durchgeführt hat. Hier bietet sich die Analogie zu einem Flugdatenrekorder an, der absturzsicher gepanzert ist und alle relevanten Informationen des Fluggeschehens aufnimmt. Er kann zwar einen Crash nicht verhindern, aber nachträglich über den Unfallhergang Auskunft geben. Analog läßt sich aus Logdateien nachträglich der Weg, speziell der Verursacher einer Infektion ermitteln, wenn diese manipulationssicher geschützt sind.

Wie schon gesagt, sind für die erfolgreiche Verbreitung von Viren keine besonderen Systemrechte erforderlich, aber man muß es Softwareanomalien auch nicht unnötig leicht machen. Großrechner verfügen meist über in der Hardware implementierte Speicherschutzsysteme. Jedes Programm kann nur die ihm zugeordneten Speicherbereiche auf festgelegte Weise manipulieren. Eine völlige Kontrolle über das System, wie sie jedes Programm unter MS-DOS inne hat, ist beim Normalbetrieb nie gegeben. Nur Programme, die unter einer besonderen Benutzerkennung gestartet werden ("Superuser" = Systemadministrator), besitzen einen privilegierten Status und können alle verfügbaren Operationen durchführen. Damit sind speicherresidente Viren, die Funktionen des Betriebssystems übernehmen, nicht oder nur sehr schwer zu realisieren.

Antivirusprogramme. Programme zur Abwehr von Softwareanomalien auf APCs arbeiten meist nach den drei Grundprinzipien Überwachung (Watcher), Überprüfung auf Muster (Scanner) und Überprüfung auf Veränderungen (Checker). Keine der letzten beiden Techniken wird bisher von konventionellen Schutzmaßnahmen abgedeckt. Watcher ahmen z.T. Schutzfunktionen auf Dateiebene nach (s. `Flushot`), in dem sie schreibenden Zugriff auf ausführbare Dateien verhindern oder zumindest den Anwender alarmieren. Die Protokollierung von Operationen (Logging, Auditing) entspricht etwa einer Überwachung, *nachdem* etwas passiert ist.

Daß auf dem Mainframesektor keine Scanner existieren, liegt daran, daß es keine bekannten Viren gibt, nach denen man suchen könnte. Mittlerweile sind einige Checker für verschiedene Systeme auf dem Markt. Die bisherigen Attacken erfolgten durch Wurmprogramme, gegen die Scanner und Checker machtlos und normale, korrekt angewandte Sicherheitsvorkehrungen ausreichend sind. Das Mittel der Zukunft gegen Manipulationen jegliche Art sind nach Cohens Auffassung Betriebssysteme, welche die Integrität der gespeicherten Daten überwachen. Dieser Ansicht schließt sich der Autor aus Überzeugung gerne an.

Alternative Konzepte. Das vorgestellte Schutzzonenkonzept geht von einer anderen Situation aus, als sie bei normalen Antivirusprogrammen gegeben ist. Beim Einsatz von APCs im Betrieb oder Rechenzentren muß der Systemverwalter wissen, daß die Benutzer zum einen nicht Fachleute für Softwareanomalien sind und zum anderen die Rechner mehr oder weniger mutwillig gefährden (s.a. 2.1.3 "Motive der Anwender"). Das Schutzsystem muß so ausgelegt sein, daß einer unabsichtlichen Verseuchung ebenso vorgebeugt wird wie einer willentlichen.

Die vorgestellten drei Schutzzonen wehren Softwareanomalien auf verschiedenen, z.T. redundanten Ebenen ab. Auf vorderster Front wird der Rechner durch SZ II und III von der Außenwelt isoliert; eine Methode, mit der kein herkömmliches Antivirusprogramm arbeitet. Weder der Start eines verseuchten Programms noch das Einspielen eines (evtl. getarnten) Virus ist möglich. Selbst wenn dies gelänge, verhindert SZ I den Start, weil das Programm dem System unbekannt oder die Integrität (durch Benutzung des gleichen Namens) verletzt ist. Letztere Überwachungsfunktion greift auch dann, wenn alle Dämme gebrochen sind und ein aktiviertes Virus Programme manipuliert hat. Diese können nicht mehr ausgeführt werden, die zweite Virusgeneration wird blockiert.

Kontrollierte Isolation und andere Schutzkonzepte schließen sich nicht aus, sondern ergänzen sich im Gegenteil hervorragend. Residente Scanner erkennen Viren in Programmen unbekannter Qualität (z.B. Neuzugänge), Watcher verhindern Schäden und Verbreitung bei der versehentlichen Aktivierung eines Virus. Checker-Funktionen sind bereits durch SZ I abgedeckt.

Kapitel 3

Systemprogrammierung unter MS-DOS

> Da zahlen Leute Tausende von Mark an Gebühren für Seminare über Softwareengineering, und anschließend fahren sie nach Hause und lassen ihre Leute in C programmieren.
>
> Niklaus Wirth zitiert in c't 4/91

Dieses Kapitel stellt nach der Theorie der Abwehr von Viren den zweiten, systemspezifischen Teil der theoretischen Grundlagen dar und befaßt sich mit der Systemprogrammierung unter MS-DOS. Es kann, soll und muß keine vollständige Beschreibung des Betriebssystems MS-DOS sein, denn diese würde den Rahmen dieses Buches sprengen. Die gebotenen Informationen reichen aus, um im folgenden Kapitel wirksame Programme zur Abwehr von Softwareanomalien erstellen zu können. Zwar wird bei der Programmentwicklung versucht, nicht auf spezifische Aktionen und Fähigkeiten von MS-DOS-Viren einzugehen. Dennoch ist es notwendig, an verschiedenen Stellen in die "Niederungen" des Betriebssystems hinabzusteigen und auch mit undokumentierten Funktionen zu arbeiten.

Voraussetzungen. Programmierkenntnisse in "C" oder einer vergleichbaren Programmiersprache werden vorausgesetzt. Grundlagen in Assembler, besonders die Fein- und Besonderheiten der Intel CPUs, werden im ersten Abschnitt dargelegt. Grundlagen der Programmierung in "C" und in Assembler vermitteln [5, 4] bzw. [2, 8, 14] und [3], um nur einige zu nennen. Dieses Kapitel beinhaltet auch kurze Übungsprogramme, welche die vermittelte Theorie veranschaulichen und zeigen, wie eine Anwendung in der Praxis aussehen kann. Teile dieser Programme sind wichtiger Bestandteil der Systembibliothek, die ihrerseits wieder Grundlage der im nächsten Kapitel zu entwickelnden Systemprogramme ist. Daher seien eigene Experimente und Tests mit den Übungsprogrammen sehr empfohlen.

Hinweise zur Programmerstellung. Bei der Entwicklung der Programme für dieses Buch wurde darauf geachtet, daß möglichst keine Abhängigkeit von einem bestimmten "C"-Compiler besteht. Der Autor hat die Programme mittels Borlands Turbo-C 2.0 und dem Turbo-Assembler erstellt. Dafür war die Verwendung spezieller Funktionen und Bezeichner erforderlich, die in den Bibliotheken dieses Compilers enthalten sind und die nicht zum ANSI-Standard für "C" gehören. Grund dafür sind Erfordernisse der Hardware wie z.B. NEAR- und FAR-Adressierung sowie direkte Aufrufe von MS-DOS-Funktionen. Insbesondere die Betriebssystemaufrufe sind von Compiler zu Compiler meist verschiedenartig implementiert. Auf der Begleitdiskette sind deshalb Quelltexte zu "C"-Funktionen enthalten, die funktional insofern identisch mit den verwendeten Turbo-C-Funktionen sind, als dies für die erstellten Programme erforderlich ist. Bis auf den Funktionsnamen, dem ein '`x`' vorangestellt wurde, und einigen Bezeichnern für MS-DOS-spezifische Datenstrukturen sind die Funktionsprototypen gleich denen in den Turbo-C `#include`-Dateien. Dazu kommen weitere Funktionen, die auch Turbo-C nicht zur Verfügung stellt, sondern neu implementiert werden mußten.

Programmierrichtlinien. Bei der Erstellung der Quelltexte kamen einige Regeln zur Anwendung, die die Lesbarkeit und Verständlichkeit des Codes erhöhen sollen.

- Alle Programme und Funktionen weisen eine einheitliche Struktur auf (Reihenfolge der Deklarationen; Format des Quelltextes und der Kommentare).
- Definitionen (Makros, symbolische Konstanten) und Typdefinitionen (einfache Typen und `struct`s) erscheinen in Großbuchstaben.
- Typdefinitionen sind durch den Anfang "`T_`" gekennzeichnet.
- Symbolische Konstanten, die für Betriebsarten stehen, beginnen mit einem Kennbuchstaben für die Funktion und "`M_`" für "Modus" (z.B. `CM_DOUBLE`: Funktion `clean`, Modus "remove double spaces"). Ähnliches gilt für Rückgabewerte, die mit "`R_`" gekennzeichnet sind.
- Jeder Anweisungsblock ist, ob notwendig oder nicht, in "`{...}`" eingefaßt, wird durch ein Semikolon beendet und um zwei Leerzeichen eingerückt.
- Die Dokumentation der Quelltexte auf Diskette erfolgt in englischer Sprache, weil Kommentare knapper und prägnanter erfolgen können und viele Fachtermini sowieso aus dem Angelsächsischen stammen.
- "`>>`" zeigt an, daß an dieser Stelle Programmcode einzufügen ist.

3.1 Grundlagen Assembler

3.1.1 Die INTEL Prozessorfamilie

Alle IBM-kompatiblen PCs sind mit Prozessoren der INTEL 80*86-Serie ausgerüstet. Die erste PC-Generation, die XTs, verwendeten den 8086 oder 8088 Prozessor, die sich nur durch die unterschiedliche Breite des Datenbusses unterscheiden. Der 16 Bit-Datenbus des 8086 machte ihn 20% schneller als den 8088 mit seinem 8 Bit-Bus. Beide CPUs

verfügen über einen 20 Bit breiten Adreßbus, können also $2^{20} = 1$ MB Speicher adressieren.

Die mit den Prozessoren 80186 (selten), 80286, 80386 SX und DX und 80486 ausgerüsteten ATs stellen die zweite Entwicklungsstufe dar. Der 24 Bit breite Adreßbus ermöglicht den Zugriff auf $2^{24} = 16$ MB Speicher. Während der 80186 nur eine erweiterte Version des 8086 ist, kamen mit dem 80286 und den folgenden Modellen völlig neue Fähigkeiten hinzu, wenn der Betrieb im *Protected Mode* erfolgt. Dazu zählen virtuelle Adressierung bis 1 GB, geschützter Speicher (*Protected Memory*) und Multitasking.

Die Prozessoren sind zueinander abwärtskompatibel, d.h. Programme, die den Befehlssatz einer bestimmten CPU verwenden, laufen auch auf späteren Versionen des Prozessors. Sie profitieren auf diese Weise zwar nicht von den neu hinzugekommenen Befehlen, laufen aber wegen dem höheren Durchsatz, der durch höhere Taktfrequenzen und Änderung der internen Struktur erreicht wird, schneller ab. Eine Neuübersetzung des Programms macht den Code durch Ausnutzung der neuen Befehle und der dadurch möglichen Optimierung meist kompakter und schneller. Um abwärtskompatibel zu sein, bieten die neueren Prozessoren neben dem Protected Mode den *Real Mode* an, in dem sich die CPU wie ein schneller 8086 verhält.

Ein Wermutstropfen: MS-DOS nutzt die Fähigkeiten, die die neuen Prozessoren bieten, nicht aus, sondern läuft aus Kompatibilitätsgründen im Real Mode ab. Durch die seit den Gründertagen unveränderte Speicherverwaltung ist für MS-DOS oberhalb von 1 MB der Speicher zu Ende und nur über Hilfskonstruktionen erreichbar. Die Verwendung eines 80386er ATs unter MS-DOS entspricht deshalb dem Fahren eines Porsches im ersten Gang mit angezogener Handbremse, aber immerhin mit durchgetretenem Gaspedal. Erst unter dem Betriebssystem OS/2 und manchen UNIX-Implementationen werden die Prozessoren richtig "ausgefahren".

3.1.2 Das Segment-Konzept

In diesem Abschnitt geht es um alles, was der Systemprogrammierer beim Umgang mit den INTEL-CPUs beachten muß und, das darf man wohl sagen, ihm das Arbeiten potentiell erschweren wird.

Die INTEL-Prozessoren sind im Real Mode, den MS-DOS verwendet, intern 16 Bit breit organisiert, was historische Gründe hat. Das bedeutet, daß die Daten- und Adreßregister die Größe eines 16 Bit-Wortes (im folgenden kurz: Wort) besitzen. Damit lassen sich 65536 verschiedene Zustände darstellen, z.B. die Zahlen von 0 bis 65535 (entspricht in "C" `unsigned int`) oder -32768 bis $+32767$ (entspricht `int`). Aus dieser Beschränkung ergibt sich ein Problem, denn im Real Mode kann insgesamt 1 MB Speicher adressiert werden, wofür, wie oben angedeutet, 20 Adreßbits erforderlich sind.

Die Lösung besteht darin, zwei 16 Bit-Register zu einem größeren Register zusammenzuschalten. Der Gedanke liegt nahe, von den so erhaltenen 32 Bits die höherwertigen 12 Bits "wegzuwerfen" und den Rest als 20 Bit-Adresse zu verwenden. Das "obere", höherwertige Wort bestimmt einen 64 kB-Block (*Segment*) innerhalb des 1 MB

Adreßraums. Das "untere", niederwertige Wort legt die Adresse oder den *Offset* innerhalb eines Segments relativ zu dessen Anfang fest. Mit diesem Verfahren erhält man 16 Segmente mit jeweils 64 kB, die an 64 kB-Grenzen beginnen, wie in Abb. 3.1 oben zeigt.

Abbildung 3.1: Mögliche und tatsächliche Segmentierung des Adreßraums

Wie wir später noch sehen werden, verwenden die meisten Programme mehr als ein Segment, um den von ihnen belegten Speicher in Code- und Datenbereiche zu strukturieren. Bei der momentanen Technik müßte ein Programm mit zwei Segmenten mind. 64 kB umfassen, weil Segmente nur an 64 kB-Grenzen beginnen können. Deshalb entschied sich INTEL dafür, alle 16 Bits des oberen Adreßworts, der *Segmentadresse* (Segmentnummer, Segment), zu nutzen. Dadurch ergibt sich ein minimaler Segmentabstand von nur noch $\frac{1024\text{ kB}}{65536\text{ Segmente}} = 16$ Bytes, d.h. ein Segment beginnt an einer durch 16 ohne Rest teilbaren Adresse. Die Einheit von 16 Bytes bezeichnet man als *Paragraph*; eine Adresse, an der ein Segment beginnen kann, als *Paragraphengrenze*. Diese neue Situation ist in Abb. 3.1 unten dargestellt. Weil die Segmentgröße von 64 kB beibehalten wird, können sich Segmente überschneiden.

Beispiel "Berechnung Basisadresse aus Segmentnummer"
Das Segment Nummer 100 beginnt an der Adresse 100_{10}[1] $*16_{10} = 1600_{10}$ oder einfacher noch in sedezimaler Schreibweise (also zur Basis 16) $0064_{16} * 0010_{16} = 0640_{16}$. Um aus der Segmentnummer die *Basisadresse* zu erhalten, wird diese um 4 Bits oder eine Sedezimalstelle nach links geschoben, was gleichbedeutend mit einer Multiplikation mit 16 ist. Dazu noch ein Beispiel:

[1] Die tiefgestellte Zahl stellt die verwendete Basis dar.

Segment $3F7E_{16}$, eine "0" anhängen ⇒ Basisadresse $3F7E0_{16}$.

Beispiel "Berechnung lineare Adresse aus Segment/Offset"
Innerhalb eines Segments erfolgt die Adressierung über den Offset, der zu der durch das Segment festgelegten Basisadresse hinzuaddiert wird. Beispiel:
Segment 0064_{16} ⇒ Basisadresse 00640_{16}; Offset $000F_{16}$
⇒ lineare Adresse = $00640_{16} + 000F_{16} = 0064F_{16}$. Besonders einfach wird der Vorgang, wenn man die Zahlen untereinander schreibt:

```
0064    Segment (um eine Sedezimalstelle nach links verschoben)
 000F   + Offset
0064F   = lineare Adresse
```

Die Beispiele zeigen, wie eine aus Segment und Offset bestehende Adresse in eine *lineare Adresse* umgeformt wird. "Linear" deshalb, weil die nächste Adresse einfach durch Erhöhen oder Erniedrigen des Adreßzählers angesprochen werden kann. Der Adreßraum erscheint fortlaufend und in einem Stück, ohne daß zwei separate Register, Segment und Offset, gleichzeitig betrachtet werden müssen. Das Verständnis des eben Dargelegten ist wesentlich für die folgenden Ausführungen und die Programmierung von IBM-kompatiblen PCs. Die Aufteilung von Adressen in Segment und Offset wird uns beim Entwurf der Schutzprogramme auf Schritt und Tritt begleiten, um nicht zu sagen: verfolgen.

Segmentregister. Die INTEL-CPUs besitzen eine Reihe von 16 Bit breiten Registern, die sich in vier Klassen einteilen lassen: Segment-, Index-, Arbeits- und Spezialregister (Tab. 3.1).

Vier Segmentregister legen die Basis von Segmenten mit bestimmten Aufgaben fest. Das `CS`-Register enthält den Beginn des Codesegments, in dem sich die Programmbefehle befinden. Sprünge und Unterprogrammaufrufe beziehen sich stets auf das `CS`-Register, das nicht direkt verändert werden kann (s.a. Abschnitt über NEAR/FAR-Adressierung). `DS`- und `ES`-Register bestimmen die Basis des Daten- bzw. Extrasegments, die beide Daten enthalten. Normalerweise liegt den Schreib- und Leseoperationen implizit immer eine Adressierung relativ zum `DS`-Register zugrunde. Spezielle Befehle, die Daten von einem Segment in ein anderes transportieren, verwenden das `ES`-Register als Basis des Zielsegments. `SS` ist der Name des Stacksegment-Registers, dessen Funktion wir später noch ausführlich behandeln werden.

Indexregister. Allein mit den Segmentregistern läßt sich noch nicht viel anfangen, denn zum Aufbau einer kompletten Adresse ist noch die Angabe des Offsets erforderlich. Dieser wird entweder direkt durch eine Zahl oder durch den Inhalt eines der Indexregister bestimmt. Die Angabe von Segment und Offset bezeichnet man als FAR-Adressierung. Bei dieser *intersegmentalen* Adressierung sind zwei Worte für die Adreßangabe erforderlich. Diese Methode ermöglicht bei Sprüngen und Unterprogrammaufrufen den Wechsel des Codesegments und damit Programme mit mehr als 64 kB Code. Für FAR-Zugriffe auf Daten muß das `DS`- oder `ES`-Register explizit umgeladen werden. Ist nur der Offset angegeben, spricht man von NEAR-Adressierung, mit der nur Adressen innerhalb eines Segments erreichbar sind (*intrasegmentale* Adressierung).

Name	Bedeutung	Hinweise
Segmentregister		Segmentinhalt
CS	Codesegment	Programmcode
SS	Stacksegment	Stack
DS	Datensegment	Daten (Quelle)
ES	Extrasegment	Daten (Ziel)
Indexregister		Standard-Basisregister
IP	Offset Programmzähler	rel. zu CS
SP	Offset Stackpointer	rel. zu SS
BP	Basiszeiger	rel. zu SS
SI	Quellindex	rel. zu DS
DI	Zielindex	rel. zu ES
Arbeitsregister		
AX	Akkumulator	
DX	Erweiterung Akkumulator	
CX	Zählregister	
BX	Vielzweck- oder Basisregister	
Spezialregister		
PSW	Programmstatuswort	enthält die Prozessor-Flags

Tabelle 3.1: Bedeutung der Register

Eine NEAR-Adresse besteht demnach aus nur einem Wort und bezieht sich stets auf ein Segmentregister. NEAR-Adressierung ist für Code und Daten möglich.

Für den Anwendungsprogrammierer stehen die Register SI (Source Index = Quellindex), DI (Destination Index = Zielindex) und BP (Basepointer = Basiszeiger) frei zur Verfügung. Besondere, von der CPU selbst verwendete Indexregister sind der Offset-Teil des Programmzählers, IP (Instruction Pointer), und des Stapelzeigers, SP (Stack Pointer). Das Paar CS:IP zeigt stets auf den Befehl, der gerade ausgeführt wird. Deshalb besteht auf IP wie auf CS aus naheliegenden Gründen kein direkter Schreibzugriff, denn dadurch würde sich der Programmablauf verändern. Bei Sprüngen und Unterprogrammaufrufen dagegen ist dies durchaus erwünscht, und deshalb verändern diese Kommandos den Programmzähler.

Einschub: Funktion des Stack. Der Stapelzeiger (engl. stackpointer) besteht aus dem Registern SS (Stack Segment) und SP (= Stack Offset) und zeigt auf die oberste Adresse des Stapels. Dieser dient der CPU zur Speicherung der Rücksprungadresse, falls das Hauptprogramm ein Unterprogramm aufruft oder durch einen Interrupt unterbrochen wird. Aber auch der Anwender kann Daten wortweise auf dem Stack deponieren und wieder zurückholen. Im weiteren Text ist mit "Stackpointer" oder "Stapelzeiger" das Registerpaar SS:SP und nicht nur der Offset-Teil SP gemeint.

Auf dem Stack werden Daten wie auf einen Kartenstapel (→ Name) abgelegt. Der Stapel wächst durch Hinzufügen und Wegnehmen einzelner Karten, wobei stets nur Zugriff auf die oberste Karte besteht. Die letzte aufgelegte Karte wird zuerst aufgenommen. Diese Methode heißt LIFO, was für "**L**ast **I**n / **F**irst **O**ut" steht. Genau so

funktionieren die Befehle `push` und `pop`, die ein Datenwort auf dem Stack ablegen oder vom Stack holen. Vorteil dieses Verfahrens ist, daß

- die tatsächliche Speicheradresse nicht bekannt sein muß.
- die CPU die Verwaltung des Stack mittels des Stapelzeigers übernimmt.
- automatisch eine Reihenfolge der Daten gegeben ist.
- kleine Speichermengen für verschiedene Zwecke vom Stack genommen werden können.

Werden auf dem Stack Werte z.B. mit dem Maschinensprachebefehl `push` abgelegt, wächst dieser nach unten in Richtung der niedrigen Adressen. Der Stackpointer wird *vor* dem Speichern des Wertes dekrementiert. In einer "C"-ähnlichen Schreibweise könnte man die Arbeitsweise des `push`-Befehls mit `*(SS:--SP) = <value>` beschreiben. Der zu dieser Operation komplementäre Befehl `pop` holt einen Wert vom Stack, und zwar immer den obersten, auf den der Stapelzeiger gerade zeigt. Die "C"-Notierung `<value> = *(SS:SP++)` verdeutlicht dies.

Das Kommando `call far` ruft ein Unterprogramm in einem anderen Codesegment auf. Der Prozessor merkt sich die Rücksprungadresse, indem er zuerst das Segment und dann den Offset des Programmzählers `CS:IP` auf dem Stack abgelegt. Dieser zeigt auf den Befehl, der dem `call`-Kommando unmittelbar folgt. Im Falle von NEAR-Aufrufen mit `call near` wird nur der Offset auf den Stack gerettet, da das Codesegment ja nicht gewechselt wird (Abb. 3.2). Die Befehle `retn` bzw. `retf` (**ret**urn from subroutine, NEAR bzw. FAR) laden die Rücksprungadresse vom Stack in den Programmzähler und bewirken so die Fortsetzung des Hauptprogramms. Das Verfahren bei einer Programmunterbrechung (engl. interrupt) mit `int` läuft analog zu einem `call far`, nur wird vor der Rücksprungadresse noch das Statusregister des Prozessors auf den Stack gerettet. Der adäquate Befehl zur Beendigung der Interrupt-Routine heißt `iret` (**ret**urn from **i**nterrupt).

Standardsegmente. Beim Zugriff auf Speicherinhalte verwendet die CPU je nach Situation standardmäßig bestimmte Segmente, damit nicht bei jedem Befehl ein Segmentregister angegeben werden muß (s.a. Tab. 3.1). Bei Lese- und Schreiboperationen bilden die Registerpaare `DS:SI` (Quelle) und `ES:DI` (Ziel) eine vollständige Adresse. Adressierung über das `BP`-Register erfolgt standardmäßig relativ zum Stacksegment. Warum das sinnvoll ist, lernen wie bei der Parameterübergabe kennen. Diese Zuordnungen lassen sich über sog. *Segment Override Prefixes* aufheben, die dem Offset durch einen Doppelpunkt getrennt vorangestellt werden (z.B. `CS:[SI]`). Dadurch ist der Zugriff auf beliebige Segmente möglich. Wie schon angesprochen, bilden außerdem `CS:IP` und `SS:SP` ein festes Paar.

Arbeitsregister. Mit Hilfe der Segment- und Indexregister kann auf vielfältige Art und Weise auf Speicherinhalte zugegriffen werden. Als Quell- und Zielregister, als Zwischenspeicher, zum Rechnen und zum Durchführen von logischen Operationen stehen vier weitere 16 Bit-*Arbeitsregister* zur Verfügung. Für Multiplikation und Division ist ausschließlich das `AX`-Register oder der "*Akkumulator*" zuständig. Datenbewegungen zwischen einer fixen Speicheradresse und dem `AX`-Register sind besonders schnell.

Abbildung 3.2: Verhalten des Stack bei `call near/far` und `int`

Bei einer 32 Bit-Division oder Multiplikation interpretiert die CPU die Kombination `DX:AX` als Registerpaar. Das `CX`-Register fungiert bei Operationen wie dem Schieben oder Rollen von Bits und bei automatischen Schleifen als Zählregister. Dem `BX`-Register ist keine bestimmte Funktion zugeordnet; es kann als einziges der Arbeitsregister auch als Indexregister verwendet werden.

Das höherwertige und das niederwertige Byte jedes Arbeitsregisters kann selbst wieder als Register angesprochen werden. So besteht z.B. das Register `AX` logisch aus den Registern `AH` für das höherwertige Byte und `AL` für das niederwertige Byte. Allgemein gilt: Das 16 Bit-Register `?X` besteht aus den beiden 8 Bit-Registern `?H` und `?L`. Eine Veränderung der Byte-Register beeinflußt das Wort-Register und umgekehrt.

Spezialregister. Das `PSW` (Programmstatuswort) enthält neun Flags, die bestimmte Zustände des Prozessors anzeigen und Betriebsarten festlegen. Interessant sind für uns nur das Carry- und das Zero-Flag, die bei manchen Betriebssystemaufrufen eine Rolle spielen.

3.1.3 Adressierung (Besonderheiten)

Dem Programmierer stehen eine Reihe von Operandentypen und Adressierungsarten zur Verfügung, die bei Operationen Quelle und Ziel spezifizieren. Diese sollen zunächst einmal vorgestellt werden, um bei der späteren Programmierung von in Assembler geschriebenen Programmodulen Verwendung zu finden. Einen guten Teil dieses Abschnitts macht die Problematik der NEAR/FAR-Adressierung aus, die sich wie ein roter Faden (ein rotes Tuch?) durch die Programmierung der INTEL-CPUs zieht. Erläutert werden mögliche Klippen und ihre Umschiffung sowie die korrekte Verwendung von Befehlen und Deklarationen.

Adressierungsarten. Die Prozessorbefehle verlangen entweder keinen, einen oder zwei Operanden. Bei zwei Operanden ist der eine je nach Befehlstyp ein Register oder eine Konstante/Adresse (Immediate- bzw. Absolute-Adressierung), der andere je nach Modus ebenfalls ein Register oder eine Speicherzelle. Zuweisungen von Speicherzelle zu Speicherzelle oder zwischen Segmentregistern sind nicht möglich und erfordern einen Umweg über z.B. ein Arbeitsregister ("Load and Store"-Architektur).

Tabelle 3.2 zeigt, wie sich die *effektive Adresse* (EA) der anzusprechenden Speicherzelle je nach Adressierungsart berechnet. Die ersten beiden Einträge stellen keine Adresse dar; der Wert der Ausdrücke wird unmittelbar verwendet. Die eckigen Klammern `[...]` bedeuten "Inhalt von". "Disp" steht für Displacement (engl.: Verrückung) und bezeichnet einen festen Wert, der zur Adresse hinzuaddiert wird.

Effektive Adresse	Bezeichnung
<Wert>	Immediate
<Register>	Register
<Disp>	Absolute
[BP] {+ <Disp>}	Indirect {Offset}
[SI] {+ <Disp>}	
[DI] {+ <Disp>}	
[BX] {+ <Disp>}	
[BX] + [SI] {+ <Disp>}	Indirect Indexed {Offset}
[BX] + [DI] {+ <Disp>}	
[BP] + [SI] {+ <Disp>}	
[BP] + [DI] {+ <Disp>}	

Tabelle 3.2: Adressierungsarten

Der vorangegangene Text war eine größere Ansammlung theoretischer CPU-Interna, die durch die folgenden Beispiele ein wenig mit Leben erfüllt werden sollen. Links steht der Befehl, wie ihn der Assembler kennt; rechts steht eine äquivalente Version in "C"-Schreibweise.

```
MOV AX, 2342h                % AX = 0x2342
MOV CL, 4h                   % CL = 0x04
MOV AX, DX                   % AX = DX
MOV AX, [74Ah]               % AX = *((WORD *)MK_FP (DS, 0x074A))
MOV AH, [BX]                 % AX = *((BYTE *)MK_FP (DS, BX))
MOV AX, [BP].42h             % AX = *((WORD *)MK_FP (SS, BP + 0x42))
MOV AX, ES:[E793]            % AX = *((WORD *)MK_FP (ES, 0xE793))
MOV [9CB4], AX               % *((WORD *)MK_FP (DS, 0x9CB4)) = AX
MOV BYTE PTR [6D6C], 42h     % *((BYTE *)MK_FP (DS, 0x6D6C)) = 0x42
```

Dazu noch ein einige Erläuterungen:

- Das Maschinensprachekommando zum Transportieren von Daten hat die Syntax `mov <Ziel>, <Quelle>`
- Das "C"-Makro `MK_FP` (**m**a**k**e **f**ar**p**ointer) erzeugt einen FAR-Zeiger vom Typ `void far *`, indem der erste Operand als Segment und der zweite als Offset interpre-

tiert wird. In den Argumenten dieses Makros tritt besonders die implizite und explizite Verwendung der Segmentregister hervor.

- Die letze Zeile zeigt die Verwendung der Deklarationen `BYTE PTR` und `WORD PTR`, die dem Assembler anzeigen, ob die effektive Adresse einen Byte- oder einen Wort-Zeiger darstellt. Allein aufgrund des Wertes kann diese Entscheidung nicht getroffen werden. Z.B. paßt der Wert 42_{16} sowohl in ein Byte als auch ein Wort.

Manche der im 4. Kapitel zu erstellenden Programme verwenden in Assembler geschriebene Teile, wenn dies unbedingt erforderlich ist. An dieser Stelle eine alphabetische Liste der Befehle zu bringen, erscheint dem Autor etwas zu trocken. Wir klären deshalb die Bedeutung der Kommandos an der Stelle, an der wir sie brauchen.

Problem: NEAR- und FAR-Adressierung. Die neueren CPUs der 80?86-Serie kennen die Schwierigkeiten mit den relativ kleinen Segmenten nicht, falls — und wohlgemerkt nur dann — der Betrieb im Protected Mode erfolgt. Hier stehen 32 Bit breite

Register zur Verfügung, mit denen ein Adreßraum von $2^{32} = 4$ GB erschlossen wird, ohne jemals das Segment zu wechseln. Das dürfte für die meisten Anwendungen ausreichen.

Aus der berechtigterweise etwas umständlich erscheinenden Adreßaufteilung im Real Mode ergeben sich einige Vor- und Nachteile, deren Verständnis für die Systemprogrammierung unter MS-DOS quasi lebensnotwendig ist. Damit kein Mißverständnis aufkommt: Die Segmentierung des Speichers ist prinzipiell wünschenswert und für Multitaskingsysteme schlicht notwendig. Speicherschutzmechanismen lassen sich nur auf diese Weise, nämlich über die Trennung der Adreßräume der einzelnen Programme und von Code, Daten und Stack verwirklichen.

Das Problem im Real Mode ist, daß die Segmentregister bei großen Programmen oder bei Zugriffen auf große Speicherbereiche ständig umgeladen werden müssen, weil der Adreßraum nicht linear über 64 kB hinaus durchadressiert werden kann. Programme, die mehr als jeweils 64 kB Speicher für Programm und/oder Daten benötigen, kommen mit einem Segment allein nicht aus und müssen daher Code- und/oder Datensegment wechseln (Abb. 3.3). Deshalb spielen die Begriffe NEAR und FAR eine große Rolle in der Programmierung der INTEL-CPUs.

Abbildung 3.3: NEAR- und FAR-Adressierung

Beispiel "Probleme bei NEAR/FAR-Aufrufen"
Ein absoluter Sprung innerhalb eines Segments (intrasegmental) benötigt nur die Angabe des Ziel-Offsets, also eines Wortes. Für einen Sprung in ein anderes Segment (intersegmental) sind zwei Worte, nämlich Segment und Offset, notwendig. Beim Aufruf eines Unterprogramms wird die Rücksprungadresse (= Adresse des dem Aufruf folgenden Befehls) auf dem Stack hinterlegt, um nach Ablauf des Unterprogramms mit der Abarbeitung des Hauptprogramms fortfahren zu können. Je nach NEAR- oder FAR-Sprung belegt die Rücksprungadresse ein oder zwei Worte auf dem Stack. Auf die Verwendung des korrekten `ret`-Befehls zur Beendigung des Unterprogramms, der in den Versionen für NEAR (`retn`) und FAR (`retf`) zu haben ist, ist unbedingt zu achten.

Ein Rücksprung mit dem falschen Exemplar hat fatale Folgen: Entweder endet der Rücksprung im Chaos und/oder der Stack wird verwüstet.

Problem: Adreßvergleich. Das Verfahren der Speichersegmentierung hat weiterhin den unangenehmen Nachteil, daß Adressen nicht ohne weiteres miteinander vergleichbar sind, wie das Beispiel in Tab. 3.3 zeigt.

Segment	Offset	lineare Adresse	Segment	Offset
0010_{16}	1000_{16}	01100_{16}	0110_{16}	0000_{16}
0088_{16}	0880_{16}	01100_{16}	0110_{16}	0000_{16}
1234_{16}	5678_{16}	$179B8_{16}$	$179B_{16}$	0008_{16}
$0FBC_{16}$	$7DF8_{16}$	$179B8_{16}$	$179B_{16}$	0008_{16}

Tabelle 3.3: Probleme beim Adreßvergleich

Obwohl Segment und Offset jeweils voneinander verschieden sind, beschreiben die Paare dennoch die gleiche lineare Adresse. Einen Ausweg bietet die *Normalisierung* der linearen Adresse an, bei der die Aufteilung zwischen Segment und Offset so vorgenommen wird, daß der Offset kleiner gleich $15_{10} = 000F_{16}$ ist. Anders gesagt, werden Segment und Offset in die zugehörige lineare Adresse umgewandelt und anschließend die niederwertigen vier Bits als Offset und die höherwertigen 16 Bits als Segmentadresse verwendet (Tab. 3.3, letzte beide Spalten).

Turbo-C bietet als Ausweg den Zeiger-Typ `huge` an, bei dem der Zeiger automatisch nach jeder Operation (Addition/Subtraktion) mit entsprechendem Aufwand an Rechenzeit normalisiert wird. Dadurch sind `huge`-Zeiger im Gegensatz zu ihren FAR-Kollegen miteinander über die logischen Operatoren `<`, `==` und `>` mit dem erwarteten Ergebnis vergleichbar.

Adreß-Arithmetik. Auch bei der Addition und Subtraktion von Adressen ist zumindest in der nicht normalisierten Form Vorsicht geboten.

Beispiel "Fehler durch Offset Über-/Unterlauf"
Ein Puffer der Größe 144 kB soll sequentiell aufsteigend bearbeitet werden. Dazu ist nach jedem Verarbeitungsschritt die (lineare) Adresse zu inkrementieren. Die Erhöhung des Offsets allein genügt dabei nicht. Nach spätestens $2^{16} - 1 = 65535$ Schritten findet ein Überlauf (*Wrap Around*) statt, bei dem der Offset auf Null zurückgesetzt wird, das Segment aber unbeeinflußt bleibt. Es werden dann Bereiche durchlaufen, die — vom fehlerhaften Zugriff einmal ganz abgesehen — u.U. nicht zum Puffer gehören (Abb. 3.4: Inkrementierung hier in 32 kB-Schritten).

Speicherbereiche von mehr als 64 kB Umfang können generell nur über Beeinflussung von Segment *und* Offset adressiert werden. Läuft der Offset über, ist ein Segmentwechsel erforderlich. Eine Möglichkeit der Implementierung dieses Verfahrens besteht darin, die Segmentnummer um eins zu erhöhen und den Offset auf 0000_{16} zu setzen, wenn der Offset nach der Inkrementierung den Wert 0010_{16} erreicht hat. Der Typ `huge` von Turbo-C führt diese Normalisierung nach adreßarithmetischen Operationen automatisch durch.

Abbildung 3.4: Fehlerhafte Adressierung bei Offsetüber- und -unterlauf

3.1.4 Interrupts

Eine besondere Art des Unterprogrammaufrufs stellen *Unterbrechungen* (engl. *interrupts*) dar. Diese werden durch Signale oder bestimmte Zustände der CPU (Hardwareinterrupt) sowie durch spezielle Maschinensprachebefehle (Softwareinterrupt) ausgelöst. Für unsere Belange ist die Betrachtung der Softwareinterrupts ausreichend, die durch den Befehl `int <nummer>` aktiviert werden.

Funktion. Die CPUs der 80*86-Serie bieten die Verwendung von 256 verschiedenen Interrupts an. Die Adresse des aufzurufenden Programms wird indirekt über die Interruptnummer bestimmt. Diese bezeichnet einen Eintrag in der Tabelle der Interruptvektoren, die ab Adresse `0000:0000` im Speicher steht und verändert werden kann. Jeder Tabelleneintrag ist ein FAR-Zeiger auf eine Serviceroutine, die den Interrupt bedient. Der Offset des Eintrags berechnet sich zu 4∗<Interruptnummer>, da die Speicherung einer FAR-Adresse vier Bytes benötigt.

Wie bei konventionellen Unterprogrammaufrufen auch muß die Stelle gespeichert werden, an die nach der Unterbrechung zurückgekehrt werden soll. Dazu rettet der Prozessor zuerst das Flagregister PSW und dann, analog zu einem `call far`, den aktuellen Programmzähler `CS:IP` auf den Stack. Anschließend verzweigt die Programmausführung zur *Interrupt Service Routine* (ISR). Das zu verwendende Rücksprungkommando heißt `iret` (**ret**urn from **i**nterrupt), das neben dem Programmzähler auch das Flagregister restauriert. Abb. 3.5 stellt noch einmal graphisch den Ablauf des Vorgangs dar.

Die Verwendung von Softwareinterrupts hat den Vorteil, daß das aufrufende Programm die Adresse der Zielroutine nicht kennen muß; die Angabe einer simplen Nummer genügt. So lassen sich z.B. alle MS-DOS-Dateifunktionen mit dem Befehl `int 21h` aufrufen. Obwohl die tatsächliche Lage der Routine im Speicher von Version zu Version des Betriebssystems verschieden ist, laufen Programme einwandfrei und ohne Anpassung ab.

Viele der über Softwareinterrupts angebotene Dienste unterscheiden noch einmal zwischen mehreren Funktionen, die meist durch den Inhalt des `AX`-Registers ausgewählt werden. So sind z.B. die mehr als 100 DOS-Funktionen allesamt über den Interrupt

Abbildung 3.5: Bearbeitung eines Software-Interrupts

21_{16} erreichbar. Beim Aufruf eines Interrupts ist zu beachten, daß die Serviceroutine möglicherweise Registerinhalte zerstört und die Bereitstellung von Datenbereichen und Parameterblöcken erfordert.

MS-DOS stellt Funktionen zum Lesen und Ändern von Interruptvektoren zur Verfügung, so daß Programme nicht direkt auf die Vektortabelle zugreifen müssen. Für die beiden Betriebssystemaufrufe finden sich bei Turbo-C die Funktionen

```
void interrupt (*getvect (int intr_num)) ()              und
void setvect (int interruptno, void interrupt (*isr) ())
```

`intr_num` und `interruptno`[2] geben die Nummer des zu lesenden bzw. schreibenden Vektors an (Tab. A.8 und A.3).

Erweiterung von ISRs. Die Funktionen einer ISR lassen sich leicht erweitern oder kontrollieren, indem man den entsprechenden Vektor auf eine eigene Serviceroutine verstellt ("umbiegt"; Abb. 3.6). Denkbare Anwendungen sind die Implementation neuer Funktionen (z.B. Netzfunktionen der Novell Netware), die Erweiterung bestehender Routinen (z.B. durch das ROM-BIOS vieler Grafikkarten) und die Kontrolle von Systemaufrufen; ein Punkt, der uns besonders interessiert. Der Abschnitt 4.5.1 "Das Interrupt/"C"-Interface" geht ausführlich auf die Probleme ein, die bei der Verwendung einer "C"-Funktion als ISR entstehen. Nach erfolgter Bearbeitung des Aufrufs übergibt die eigene ISR die Kontrolle an die Originalroutine. Auf diese Weise können sich viele Programme in die Bedienung eines Interrupts einklinken, ohne das andere Programme, die diesen Service aufrufen, etwas davon bemerken.

[2] Es lebe die Konsistenz...

Abbildung 3.6: Einklinken in Interrupts

3.2 Grundlagen Hochsprachen

Dieser Abschnitt behandelt die Realisierung von Hochsprachenkonstrukten in "C" auf Maschinenspracheebene. Dazu gehören Funktionsaufrufe, die Parameterübergabe sowie die Verwendung globaler und lokaler Variablen. Notwendig wird dieser Abstieg auf die tiefste Ebene der Programmierung durch die Verwendung eigener, in Maschinensprache geschriebener Funktionen und Interfaces. Diese sollen mit in "C" implementierten Programmodulen zusammenarbeiten. Beispiele zeigen, wie der Compiler Quelltext in Assemblerprogramme umsetzt. Daraus ergeben sich an den Aufbau der eigenen Assemblermodule bestimmte Anforderungen, die Bestandteil des Abschnitts 3.2.4 "Der Bindeprozeß" sind.

3.2.1 Speichermodelle

Theoretisch ließen sich viele Probleme der Segmentierung vermeiden, würde man stets FAR-Adressierung in der normalisierten Form verwenden. Das hat aber den Nachteil, daß der Code durch die permanenten Segmentwechsel aufwendiger und undurchschaubarer wird sowie mehr Platz und Zeit verbraucht. Microsoft schlägt daher die Benutzung bestimmter *Speichermodelle* vor, um je nach Größe von Code- und Datenbereichen einen möglichst geringen Aufwand zu treiben (Tab. 3.4).

Das Speichermodell `TINY` ist eine Sonderform, die dem Programmtyp `com` unter MS-DOS besonders angepaßt ist. Code, Daten und Stack befinden sich in einem einzigen Segment, d.h. das Programm muß alles in allem kleiner als 64 kB sein. Ebenfalls ein Spezialfall ist der Typ `HUGE`, bei dem durch Zeigernormalisierung Datensegmente mit mehr als 64 kB zulässig sind.

Für unsere Belange reicht das Modell `SMALL` völlig aus; Code- und Datenbereich sind jeweils kleiner als 64 kB. Für ein Datenbankprogramm zum Beispiel, welches mit relativ wenig Code viele Daten verwaltet, wäre das Modell `COMPACT` angemessen. Aus

Speichermodell	Code-Segmente	Datensegmente
(TINY)	1 (NEAR)	1 (Code- und Datensegment identisch)
SMALL	1 (NEAR)	1 (NEAR)
MEDIUM	n (FAR)	1 (NEAR)
COMPACT	1 (NEAR)	n (FAR)
LARGE	n (FAR)	n (FAR)
(HUGE)	n (FAR)	n (FAR; über `huge` auch größer als 64 kB)

Tabelle 3.4: Speichermodelle

der Anzahl der Segmente ergibt sich, ob NEAR- (ein Segment) oder FAR-Adressierung (n Segmente) erforderlich ist.

3.2.2 Lokale Variablen

Viele Funktionen verwenden lokale Variablen, die nur innerhalb der Funktion existieren und von außerhalb nicht zugänglich sind. Beim Eintritt in die Funktion wird für die lokalen Variablen Speicher reserviert, der beim Verlassen wieder freigegeben wird. Dazu bedient sich der Compiler des Stack.

Beispiel "Lokale Variablen"
Das folgende "C"-Programm wurde mit dem Turbo-C-Compiler in Maschinensprache übersetzt (Speichermodell `SMALL`).

```
void main (void)
{ int a, b, c;

  a = 2; b = 3;
  c = a + b;
};
```

Betrachten wir den Assemblerauszug des Beispielprogramms. Zunächst wird `BP` auf den Stack gerettet und `SP` in `BP` gespeichert. Die drei `int`-Werte benötigen insgesamt 6 Bytes lokalen Speicher. Durch den Abzug der Zahl 6 vom Stapelzeiger wird der Stapel um diese Menge Bytes aufgestockt, der Speicher wurde "reserviert". Am Ende der Funktion erhält das Register `SP` seinen alten Wert, den es vor der Reservierung hatte, zurück. Damit wird der Speicher, den die lokalen Variablen belegten, wieder freigegeben.

Hinweis: Vom Stack darf immer nur eine gerade Anzahl Bytes angefordert werden, weil die CPU (bei Unterprogrammaufrufen etc.) sowie die Befehle `push` und `pop` wortweise arbeiten! Da `BP` den Zustand von `SP` *vor* der Reservierung enthält, liegen die lokalen Variablen "weiter oben" auf dem Stack, d.h. auf niedrigeren Adressen. Dies drückt in der Adressierung der Variablen relativ zu `BP` aus: Das Displacement ist stets negativ.

```
_main   proc    near
        push    bp                      ; BP und SP retten
        mov     bp,sp
```

```
        sub     sp,6                    ; Speicher reservieren

        mov     word ptr [bp-6],2       ; "a = 2;"
        mov     word ptr [bp-4],3       ; "b = 3;"
        mov     ax,word ptr [bp-6]      ; "AX = a";
        add     ax,word ptr [bp-4]      ; "AX += b"
        mov     word ptr [bp-2],ax      ; "c = AX"

        mov     sp,bp                   ; SP und BP restaurieren
        pop     bp
        ret
_main   endp
```

3.2.3 Parameterübergabe

Meist läßt sich ein Programm vereinfachen und übersichtlicher gestalten, wenn man für häufig benötigte Funktionen Unterprogramme vorsieht. In der Regel tauschen aufrufendes (engl. *caller*) und aufgerufenes Programm (engl. *callee*) Informationen miteinander aus. Das Hauptprogramm übergibt die Daten an das Unterprogramm, das diese evtl. unter der Verwendung von Hilfsdaten bearbeitet und die Ergebnisse der Operation an das Hauptprogramm zurückgibt. Damit ein Austausch erfolgen kann, muß die Information für beide Programme gleichermaßen zugänglich sein. Dazu bieten sich Prozessorregister und gemeinsam genutzte Speicherbereiche (engl. shared oder common memory) an.

Parameter werden bei MS-DOS- und BIOS-Aufrufen über CPU-Register übergeben. Diese können als Zeiger auf größere und komplexere Datenstrukturen verweisen. Hochsprachen wie "C" und Pascal verwenden den Stack als Zwischenablage für lokale Variablen und die Parameterübergabe sowie ein bestimmtes Registerpaar für die Rückgabe von Funktionswerten.

Call by Value/Reference. Als Argumente bezeichnet man die konkreten an die Funktion übergebenen Werte, als Parameter die Variablen in der Funktionsdeklaration, die diese aufnehmen. "*Call by Value*" (Aufruf durch Wertübergabe) bedeutet, daß der aktuelle Wert eines Arguments übergeben wird. Die aufgerufene Funktion kann den Parameter, der lokal zur Funktion ist, verändern, nicht aber das Argument in der aufrufenden Funktion. Bei einem "*Call by Reference*" (Aufruf durch Adreßübergabe) wird die Adresse des Arguments an den korrespondierenden Parameter übergeben. Über diese kann die Funktion das Argument manipulieren.

Beispiel "Call by Value/Call by Reference ("C")"
Betrachten wir das folgende Beispielprogramm, das uns in z.T. abgewandelter Form in diesem Kapitel noch öfters begegnen wird. Eine Prozedur mit den Parametern `x`, `y` und `z` wird mit den Argumenten `a`, `b` und `c` aufgerufen. Nach der Rückkehr soll `c` die Summe von `a` und `b` enthalten.

Ein "Call by Value" (`add_cbv`) bringt nicht das gewünschte Ergebnis. Zwar wird mit `z = x + y` die Summe von `x` und `y` dem Parameter `z` zugewiesen, doch das Ergebnis geht beim Rücksprung mit der Freigabe des Speichers für die lokalen Variablen verloren; das Argument `c` wird nicht beeinflußt. Nur wenn `c` "by Reference" (`add_cbr`),

also die Adresse von `c`, mit `&c` übergeben wird, kann die Funktion `c` beeinflussen. Die Ausführung von `*z = x + y` wirkt dann wie `c = x + y`.

```
void main (void)
{ int a, b, c;

  a = 2; b = 3; c = 23;
  add_cbv (a, b, c);
  printf ("call by value:     %d\n", c);
  add_cbr (a, b, &c);
  printf ("call by reference: %d\n", c);
};

void add_cbv (int x, int y, int z)
{ z = x + y;
};

void add_cbr (int x, int y, int *z)
{ *z = x + y;
};
```

Die "C" Calling Convention. Die aufrufende Funktion legt Werte sowie Zeiger auf Werte, Strukturen und Funktionen auf dem Stack ab. Die dabei verwendete "*C Calling Convention*" besagt, daß die Werte in umgekehrter Folge auf den Stack gebracht werden, in der sie im Quelltext erscheinen. Für Werte, die zwei Worte umfassen, gilt, daß der höherwertige Teil zuerst abgelegt wird.

Durch diese Prozedur liegt das erste Argument im Stack an — von der Adresse her — unterster Stelle, das letzte am weitesten oben. Damit liegen die Argumente so im Speicher, wie es der Quelltext vermuten läßt. Doppelworte haben ebenfalls ihre korrekte Form: niederwertiges Wort gefolgt vom höherwertigen Wort (Abb. 3.7). Die Zahlen rechts von der Darstellung des Stack (links) geben das Displacement an, das für den Zugriff auf die Parameter zu `BP` zu addieren ist (z.B. 08_{16} für `long`). Die Assembler-Struktur rechts definiert symbolische Konstanten, welche als Displacements benutzt werden können.

Die aufgerufene Funktion findet auf dem Stack an oberster (von der Adresse her unterster) Position zunächst die Rücksprungadresse und dann die Argumente vor. Beim Aussprung aus der Funktion wird die Rücksprungadresse wieder vom Stack in den Programmzähler geladen und das Hauptprogramm muß den für die Übergabe benutzten Stackbereich wieder freigeben. Theoretisch könnte man dazu `pop`-Befehle verwenden; schneller und einfacher geht es, wenn man die Anzahl der auf dem Stack deponierten Bytes zum Stackpointer addiert[3].

Bei dem ganzen Verfahren tritt die Unterscheidung in NEAR- und FAR-Zeiger wieder äußerst störend in Erscheinung. Je nach verwendetem Speichermodell müssen Code und Daten NEAR oder FAR adressiert werden. Dies wirkt sich auf die Parameterübergabe und den zu verwendenden `ret`-Befehl aus. Abhängig davon, ob der Sprung zur Unterroutine intra- oder intersegmental erfolgt, liegen die Argumente an unterschiedlicher Position relativ zum Stackpointer auf dem Stack, weil die Rücksprungadresse ein oder zwei Bytes belegt. Dies impliziert, daß es mind. zwei Versionen einer Bibliotheksfunktion geben muß. Die Situation verschlechtert sich weiter, wenn man bedenkt,

[3] Wichtig: Für jedes `push` *zwei* Bytes.

Abbildung 3.7: Parameterübergabe via Stack

daß bei "Call by Reference" je nach Speichermodell NEAR- oder FAR-Zeiger auf Daten übergeben werden. Damit erreicht die Zahl der Kombinationen bereits vier.

Beispiel "Call by Value/Call by Reference (Assembler)"
Betrachten noch einmal das eben besprochene Programm und seine Umsetzung durch den Compiler in Assembler-Code unter Verwendung der Speichermodelle `SMALL` und `LARGE`. Zur Erinnerung: Im Modell `SMALL` befinden sich Code und Daten in je einem Segment, die über die Segmentregister `CS` bzw. `DS` und `ES` angesprochen werden. Der Inhalt dieser Register muß nie verändert werden. Das `LARGE`-Modell hingegen verwendet sowohl mehrere Code- als auch Datensegmente. Zwischen Codesegmenten wird durch FAR-Sprünge, die implizit das `CS`-Register verändern, umgeschaltet. Verschiedene Datensegmente werden explizit durch die Umbelegung des `DS`- und `ES`-Registers ausgewählt.

```
void main (void)
{ int a, b, c;

  a = 2; b = 3; c = 23;
  a = add_cbr (a, b, &c);
};

int add_cbr (int x, int y, int *z)
{ return (*z = x + y);
};
```

Nun zum Ablauf des eigentlichen Programms. Der vom Linker eingebundene *Startup*-Code des "C"-Compilers ruft nach einigen Initialisierungen die Funktion mit dem Namen `_main` auf, bei der ja die Ausführung aller "C"-Programme beginnt. Es heißt tatsächlich `_main` und nicht `main`, weil der Compiler jedem im Quelltext definierten Symbol einen Unterstrich voranstellt. Zunächst wird Platz für die vier lokalen `int`-Werte `a`, `b`, `c` und `d` reserviert und die Variablen vorbelegt.

Für den Aufruf der Funktion `add_cbr` werden die Adresse von `c` sowie die Werte von `b` und `a` in dieser Reihenfolge, die umgekehrt zu der im Quelltext ist, auf dem Stack abgelegt. Der Befehl `lea` (**l**oad **e**ffective **a**ddress) bewirkt, daß `AX` nicht der Inhalt, sondern die Adresse des rechten Operanden zugewiesen wird. Hier zeigen sich die ersten Unterschiede zwischen `LARGE`- und `SMALL`-Version im Code. `_main` selbst ist als FAR-Funktion deklariert und muß zusätzlich zum Offset der Adresse von `c` auch das Segment (`SS`) angeben. Der Aufruf von `_add_cbr` erfolgt im Unterschied zum `SMALL`-Modell mit einem FAR-`call`. Dies wiederum bedingt, daß ein Wort mehr als der Offset, nämlich das aktuelle Codesegment, als Rücksprungadresse auf den Stack gebracht wird.

```
; memory model SMALL (gekuerzt)
_main   proc    near
; Retten BP, SP; Reservierung und Vorbelegung von A, B und C
        lea     ax,word ptr [BP-2]
        push    ax                      ; (near-)Adresse von C auf Stack
        push    word ptr [BP-4]
        push    word ptr [BP-6]         ; Werte von B und A auf Stack
        call    near ptr _add_cbr       ; addieren
        add     sp,6                    ; Stack bereinigen
        mov     word ptr [BP-6],ax      ; Ergebnis an A zuweisen
; Restaurieren SP, BP; return
_main   endp

; memory model LARGE (gekuerzt)
_main   proc    far
; Retten BP, SP; Reservierung und Vorbelegung von A, B und C
        push    ss
        lea     ax,word ptr [BP-2]      ; (far-)Adresse von C auf Stack
        push    ax
        push    word ptr [BP-4]
        push    word ptr [BP-6]
        call    far ptr _add_cbr
        add     sp,8                    ; ein Wort mehr zu beseitigen!
; Zuweisung an A; Restaurieren SP, BP; return
_main   endp
```

Die Adressierung der Parameter in der aufgerufenen Funktion erfolgt relativ zum vorgefundenen Wert des Stackpointers `SP`, der in das `BP`-Register übertragen wird. Die Parameter liegen zeitlich vor der Rücksprungadresse, also räumlich gesehen darüber (im Sinne von höher liegenden Adressen). Der Zugriff auf die Parameter muß daher durch Addition eines bestimmten Wertes erfolgen, der sich aus der Größe der Rücksprungadresse und der Position der Parameter ergibt.

Die Beispielprogramme zeigen, daß sich bei der `LARGE`-Version der Wert des `BP`-Registers um ein Wort (zwei Bytes) nach unten verschiebt und deshalb die Offsets relativ zu `BP` um zwei Bytes größer ausfallen. Zu beachten ist außerdem der FAR-Zugriff auf `c` über `*z`. Der Befehl `les` (**l**oad **ES** and register) lädt das Registerpaar `ES:BX` mit dem spezifizierten Doppelwort. Das Segment Override Prefix "`ES:`" ist notwendig, weil sonst als Weglaßwert das `DS`-Register Verwendung fände. Vergleichen Sie noch einmal beide Versionen miteinander und machen Sie sich die Bedeutung jedes Unterschieds klar.

```
; memory model SMALL (gekuerzt)
_add_cbr        proc    near
; Retten von BP
```

```
        mov     ax,word ptr [BP+4]      ; "AX = x"
        add     ax,word ptr [BP+6]      ; "AX += y"
        mov     bx,word ptr [BP+8]      ; "BX = z = &c"
        mov     word ptr [bx],ax        ; "*z = c = AX"
; Wiederherstellen BP; return
_add_cbr        endp

; memory model LARGE (gekuerzt)
_add_cbr        proc    far
; Retten von BP
        mov     ax,word ptr [BP+6]      ; alle Parameter sind um ein
        add     ax,word ptr [BP+8]      ; Wort nach oben verschoben
                                        ; (bzw. BP um ein Wort nach
                                        ; unten)
        les     bx,dword ptr [BP+10]    ; (far-)Adresse von c laden
        mov     word ptr es:[bx],ax     ; Ergebnis an *z = c zuweisen
; Wiederherstellen BP; return
_add_cbr        endp
```

Für selbstentwickelte Programmodule in Assembler ist die Kenntnis dieser Feinheiten unbedingt notwendig, weil sonst das resultierende Programm bestenfalls nicht wie erwünscht funktioniert, normalerweise wohl aber abstürzt. Ein Beispiel für ein einfaches, hausgemachtes Assemblermodul werden wir im nächsten Abschnitt erarbeiten.

Rückgabe von Funktionswerten. Funktionen, die einen einzelnen Wert zurückgeben, tun dies über das Register `AX` oder das Registerpaar `DS:AX`, wobei die Belegung je nach Typ des Wertes differiert (Tab. 3.5). Der Inhalt nicht verwendeter Teile der Register ist undefiniert.

"C"-Typ	Länge	Register
char	`BYTE`	`AL`
short, int, unsigned, enum, NEAR-Zeiger	`WORD`	`AX`
long, FAR-Zeiger	`DWORD`	`DS:AX`

Tabelle 3.5: Werterückgabe bei "C"-Funktionen

Fließkommazahlen werden über das oberste Element des Stackregisters des mathematischen Coprozessors oder seiner Softwareemulation zurückgegeben. Strukturen einer Länge von mehr als vier Bytes werden als Zeiger auf diese zurückgeliefert; bis zwei Bytes wird direkt das `AX`-Register, bis vier Bytes das Registerpaar `DX:AX` verwendet.

Dimensionierung des Stack. Wichtig ist die korrekte Dimensionierung des Stack-Bereichs. Er muß groß genug sein, um lokale Variablen, Übergabeparameter und Rücksprungadressen aufnehmen zu können und gleichzeitig möglichst klein sein, damit dem Hauptprogramm genügend Speicher zur Verfügung steht. Rekursive Funktionen, die sich selbst vielleicht hundertfach aufrufen, verdienen besondere Beachtung. Bei jedem Eintritt in die Routine werden lokale Variablen angelegt und Parameter übergeben. Bei einem unkontrollierten Stacküberlauf werden u.U. Teile des Programms und der Daten überschrieben oder das Programm überschreibt Teile des Stack. Beides kann zu einem Programmabsturz führen.

3.2.4 Der Bindeprozeß

Programmodule. Programme bestehen oft nicht nur aus einem einzigen Block, sondern aus verschiedenen Modulen. Diese können als Quelltext mit `#include` oder als bereits übersetzte *Objektdateien* eingebunden werden. Diese tragen die Endung `obj` und stellen das Endprodukt eines Kompilierungslaufs dar. Das *Bindeprogramm* (engl. *linker*) bindet die einzelnen Objektdateien zu einem ablauffähigen Programm zusammen. Grund für diesen Zwischenschritt zwischen Quelltext und fertigem Programm ist, daß auf diese Weise häufig verwendete Funktionen in einer Bibliothek zusammengestellt werden können. Durch die bereits erfolgte Übersetzung läßt sich bei der Programmgenerierung erheblich Zeit sparen. Außerdem wird die Wartung von einzelnen Funktionen vereinfacht. Nur der Quelltext des Moduls, das die zu ändernde Funktion enthält, ist neu zu kompilieren. Alle Programme, die sich dieser Funktion bedienen, müssen lediglich neu gebunden werden; ein Vorgang, der automatisch erfolgen kann. Die manuelle Änderung jedes Quelltextes und die zeitraubende Rekompilierung entfällt.

Bibliotheken. In der Praxis faßt man z.B. Module mit Funktionen zur Ein-/Ausgabe und mathematische Operationen in *Bibliotheken* (engl. libraries) zusammen, die die Endung `lib` tragen. Die Verwendung von Funktionsbibliotheken bewirkt mehr Übersichtlichkeit und erspart das zeitaufwendige Öffnen und Schließen vieler separater Moduldateien. Aus dem Aufbau der Speichermodelle ist ersichtlich, daß jedes Modell von `SMALL` bis `LARGE` eine andere Kombination von NEAR- und FAR-Adressierung für Code und Daten erfordert. Deshalb existieren fünf verschiedene Turbo-C-Bibliotheken, die je nach Speichermodell `cs.lib`, `cm.lib`, `cc.lib`, `cl.lib` und `ch.lib` heißen. Im Modell `TINY` findet die `SMALL`-Bibliothek Verwendung; `HUGE` ist von `LARGE` wegen der Verwaltung von Datensegmenten mit mehr als 64 kB verschieden.

Jede Bibliothek enthält eine Reihe von Objektdateien, die Funktionen und Variablen zur Verfügung stellen oder/und selbst wieder benötigen. Die Deklaration eines Symbols als `public` (engl.: öffentlich) stellt dieses anderen Modulen zur Verfügung. Benötigte Symbole sind mit dem Schlüsselwort `extern` versehen. Es handelt sich hierbei um sog. *offene Bezüge*. Aufgabe eines Linkers ist es nun, fehlende Bezüge zwischen Modulen herzustellen. Dazu wird dem Linker eine Liste von Einzelmodulen und Biblioteksdateien übergeben, die dieser zu einem lauffähigen Programm verbinden soll. Die Reihenfolge der Module bleibt dabei erhalten und ist u.U. von wesentlicher Bedeutung für die korrekte Funktion des Programms.

Startmodul. Unter Turbo-C enthält das erste Modul jedes Programms den "Startup Code", der je nach Speichermodell `c0t.obj` bis `c0h.obj` heißt. Hier werden Speicherreservierung für Stack- und Heapbereich vorgenommen, diverse Variablen, Speicherbereiche, Emulatoren und Bildschirmadapter initialisiert, die Programmgröße an die Notwendigkeiten angepaßt, Register vorbelegt und schließlich das Hauptprogramm `_main` wie ein Unterprogramm aufgerufen. Nach Ende des Hauptprogramms erfolgt der Rücksprung zum Startup Code, wo eine Nachbehandlung und der Aussprung zum aufrufenden Programm (i.d.R. `command.com`) erfolgt. Hinweis: Der "C"-Compiler stellt intern jedem im Quelltext benutzten Symbol automatisch einen Unterstrich voran. Die für den Anwender zugänglichen Symbole der Turbo-C-Bibliotheken beginnen deshalb alle mit "_".

Dem Startup Code folgen die Programmodule des Anwenders, die in einer beliebigen Sprache verfaßt und kompiliert werden können. So lassen sich z.B. "C"- und Assemblermodule miteinander zu einem Programm kombinieren, weil das Format der Objektdateien genormt ist. Auf diese Weise können die Stärken verschiedener Programmiersprachen miteinander kombiniert werden. Zu den Anwendermodulen kommen noch eine oder mehrere Bibliotheken, die zum "C"-Compiler gehören oder von Fremdherstellern oder aus eigener Produktion stammen.

Segmentdeklarationen. Beim Bindevorgang orientiert sich der Linker an Informationen, die in den Objektdateien enthalten sind. Darunter finden sich Daten darüber, welche Symbole exportiert und welche importiert werden sowie welche logische Segmente zusammen in welches physikalische Segment gehören. Wichtig: Logische Segmente mit gleichem Namen oder Segmente, die der gleichen Gruppe angehören, kommen in das gleiche physikalische Segment und sind daher über NEAR-Adressierung erreichbar.

Microsoft sieht für die Benennung von Code-, Daten- und Stacksegmenten bestimmte Namen und Attribute vor, die fast alle Hochsprachen-Compiler für MS-DOS bei der Zwischenübersetzung in Assembler und/oder Erstellung von Objektdateien verwenden (Tab. 3.6 [8]). Für den reinen Assemblerprogrammierer haben diese Bezeichnungen keine Bedeutung. Sollen aber Assemblermodule von Hochsprachenprogrammen aus aufgerufen werden oder umgekehrt, ist die korrekte Deklaration des Segmente für den Bindeprozeß unbedingt notwendig. Das hängt damit zusammen, daß

- der Startup Code als erstes Modul die Reihenfolge der Segmente bestimmt und
- aufgrund dieser Reihenfolge der Platzbedarf des Programms bestimmt wird sowie
- bei der Programmierung der Bibliotheksfunktionen bestimmte Bedingungen definiert wurden, an die sich fremde Module halten müssen.

Beispiel: "Deklarationen im SMALL/LARGE-Modell"
Um die Auswirkungen verschiedener Speichermodelle auf das Ergebnis der Übersetzung beobachten zu können, wurde das oben bereits angesprochene "C"-Programm einmal im Modell `SMALL` und einmal in `LARGE` kompiliert. Wir betrachten nur diesmal nicht den Code, sondern die Segmentdeklarationen.

Das Resultat: Die beiden Programmversionen unterscheiden sich bereits im Kopf des Quelltextes. Die `SMALL`-Version verwendet `_TEXT` als Namen für das Codesegment. Da alle anderen Codeteile, die in der Bibliothek `cs.lib` enthalten sind, den gleichen Namen verwenden, legt der Linker beim Binden den Code in einem einzigen Segment dieses Namens ab. Dadurch ist die NEAR-Adressierbarkeit gewährleistet, die sich in der Deklaration der Funktionen `_main` und `_add_cbr` als "NEAR" ausdrückt. Die in der `DGROUP` (Gruppe der NEAR-Daten) enthaltenen Segmente (hier: `_DATA` und `_STACK`) werden ebenfalls zu einem Segment verbunden und teilen sich die verfügbaren 64 kB. Das Stacksegment taucht im Quelltext nicht explizit auf, weil dieses und dessen Zugehörigkeit zur `DGROUP` durch den Startup Code definiert ist.

Speichermodell/Inhalt	Segmentdeklaration
small	
code	_TEXT WORD PUBLIC 'CODE'
data	_DATA WORD PUBLIC 'DATA' (DGROUP)
stack	PARA STACK 'STACK' (DGROUP)
medium	
code	{<Modulname>_TEXT WORD PUBLIC 'CODE'}^{n}
data	_DATA WORD PUBLIC 'DATA' (DGROUP)
stack	PARA STACK 'STACK' (DGROUP)
compact	
code	_TEXT WORD PUBLIC 'CODE'
data	{<Name> PARA PRIVATE 'FAR_DATA'}^{m}
data	_DATA WORD PUBLIC 'DATA' (DGROUP)
stack	PARA STACK 'STACK' (DGROUP)
large	
code	{<Modulname>_TEXT WORD PUBLIC 'CODE'}^{n}
data	{<Name> PARA PRIVATE 'FAR_DATA'}^{m}
data	_DATA WORD PUBLIC 'DATA' (DGROUP)
stack	PARA STACK 'STACK' (DGROUP)

Tabelle 3.6: Namen und Deklarationen von Segmenten

```
; memory model SMALL
_TEXT   segment byte public 'CODE'
DGROUP  group   _DATA
        assume  cs:_TEXT,ds:DGROUP,ss:DGROUP

_main   proc    near
; Code
_main   endp

_add_cbr        proc    near
; Code
_add_cbr        endp
_TEXT   ends

        public  _add_cbr
        public  _main
end
```

Die LARGE-Version verwendet für das Codesegment einen eigenen Bezeichner, der sich aus dem Namen des Moduls (hier: ex2) und der Endung _TEXT herleitet. Damit erhält jeder Codeteil aus jedem Modul sein eigenes Codesegment und ist nur über FAR-Adressierung erreichbar. Der Stack ist nicht mehr Bestandteil der DGROUP, sondern in einem eigenen Segment untergebracht. Aus diesem Grund sind auch DS und SS voneinander verschieden.

```
; memory model LARGE (gekuerzt)
EX2_TEXT        segment byte public 'CODE'
DGROUP  group   _DATA
        assume  cs:EX2_TEXT,ds:DGROUP
```

```
_main   proc    far
; Code
_main   endp

_add_cbr        proc    far
; Code
_add_cbr        endp
EX2_TEXT        ends
; Export von _add_cbr und _main
end
```

Beide Programmversionen exportieren durch Deklaration als `PUBLIC` die Symbole `_main` (für den Startup Code) und `_add_cbr`. Das gilt übrigens für alle Funktionen und globale Variablen eines "C"-Programms. Das eben vorgestellte Beispiel kommt ohne externe Referenzen aus und benötigt deshalb keine Bibliotheksfunktionen. Wie sieht die Sache bei der Verknüpfung mehrerer Programmodule aus?

Beispiel "Modul-Import/Export"
Das in Turbo-C geschriebene Programm `cd.c` verwendet die (non-ANSI) Funktion `chdir`, um das aktuelle Dateiverzeichnis zu wechseln. Der Compiler bemerkt bei der Übersetzung, daß das Programm das Symbol `_chdir` zwar benötigt, aber nicht selbst definiert. In `cd.obj` wird der Aufruf von `_chdir` im Code als offener Bezug markiert und das Symbol `_chdir` als `extern` deklariert (Auszug aus Ausgabe von "`tdump cd.obj`"):

```
00012F EXTDEF 1 : '_chdir'              Type: 0
```

Damit sind alle Übersetzungsvorgänge beendet, denn das Startmodul und die Bibliotheken liegen bereits fertig kompiliert vor. Dem Linker werden die Namen des Startup-Moduls `c0s.obj` (Modell `SMALL`), des Programmoduls `cd.obj` und der Bibliotheksdatei `cs.lib` als Parameter übergeben. Bei der Suche nach einem Modul, welches das Symbol `_chdir` exportiert, stößt der Linker in `cs.lib` auf `chdir.obj`. Dieses Modul exportiert die Symbole `_chdir`, `_getdisk` und `_setdisk` und importiert `__IOERROR`:

```
; Auszug aus CHDIR.OBJ in CS.LIB
00006A EXTDEF 1 : '__IOERROR'           Type: 0
000079 PUBDEF  '_chdir'                 Segment: _TEXT:0000
000089 PUBDEF  '_getdisk'               Segment: _TEXT:0018
00009B PUBDEF  '_setdisk'               Segment: _TEXT:0021
```

`ioerror.obj` wiederum exportiert `__IOERROR`, `__dosErrorToSV` und `__doserrno` und benötigt `_errno`:

```
; Auszug aus IOERROR.OBJ in CS.LIB
00006C EXTDEF 1 : '_errno'              Type: 0
000078 PUBDEF  '__IOERROR'              Segment: _TEXT:0000
00008B PUBDEF  '__dosErrorToSV'         Segment: _DATA:0002
0000A3 PUBDEF  '__doserrno'             Segment: _DATA:0000
```

`_errno` wird zum Glück von `c0s.obj` exportiert, der Kreis schließt sich. Das Startmodul `c0s.obj` exportiert und importiert noch eine ganze Reihe von Symbolen außer `_errno` und `_main`. Diese sind hier aber nicht von Bedeutung und werden deshalb weggelassen.

Der Linker fügt alle benötigten Module zusammen und weiß daher, an welchen Adressen welche Programmteile und Daten stehen. Nun werden die offenen Bezüge

durch Eintragen der korrekten Adressen aufgelöst. Durch die einheitliche Benennung der Code- und Datensegmente mit `_TEXT` bzw. `_DATA` werden alle Codeteile und alle Daten in jeweils einem Segment untergebracht. Da `CS` Basis des Codesegments und `DS` Basis des Datensegments ist, können alle Symbole innerhalb eines Segments über NEAR-Adressierung erreicht werden — genau das fordert das Speichermodell `SMALL`.

Beispiel "Fehler beim Linkprozeß"
In den Speichermodellen `MEDIUM`, `LARGE` und `HUGE` spielt der Name des Codesegments keine Rolle, denn mehrere verschiedene Codesegmente sind zulässig. Mit FAR-Sprüngen, die der Compiler dann standardmäßig verwendet, wird das Codesegment gewechselt; sowohl die Register `CS` als auch `IP` verändern sich. Im Fall des `SMALL`- oder `COMPACT`-Modells generiert der "C"-Compiler NEAR-Aufrufe, weil er weiß, daß sich der Code in einem einzigen Segment mit dem Namen `_TEXT` befindet.

Wenn externe Module einen anderen Namen für ihr Codesegment verwenden, führen Unterprogrammaufrufe zwischen verschiedenen Modulen beim Binden zu einem "Fixup Overflow"-Fehler. Der Linker nimmt aufgrund der verschiedenen Segmentnamen an, daß mehrere Codesegmente existieren, zwischen denen er nur mit FAR-Aufrufen wechseln kann. Die tatsächliche Größe des Programms spielt dabei keine Rolle, allein die Deklaration ist ausschlaggebend. Andererseits verlangt der Code des "C"-Moduls, daß die externen Routinen über NEAR-Sprünge aufgerufen werden, sich also im gleichen Segment befinden. Da mit NEAR-Aufrufen kein Segmentwechsel möglich ist, bricht der Linker mit der genannten Fehlermeldung ab.

Beispiel "Einbinden eines Assembler-Moduls"
Die "Einführung in Assembler" ist angefüllt mit theoretischen Einzelheiten, die in diesem Beispiel zu einem praktischen Ganzen verknüpft werden sollen. Die Aufgabe: Es ist eine neue "C"-Funktion `rol` zu erstellen, die zwei Parameter `x` und `n` erwartet und als Ergebnis den `n` mal nach links gerollten Wert von `x` ausgibt. "Rollen" bedeutet bitweises "schieben" ("C"-Operator `<<`), nur daß Bits, die links "herausfallen", rechts wieder "eingeschoben" werden. Zu Demonstrationszwecken wird `x` "by reference" übergeben, um den Umgang mit dieser Art Parameter zu verdeutlichen.

Die Bedingungen:

- Funktionsprototyp `WORD rol (WORD *x, BYTE n);`
- Implementation in Assembler
- Speichermodell `SMALL` → NEAR-Funktion, NEAR-Daten; Name des Codesegments `_TEXT`

Alle Deklarationen mit `DGROUP` und `_DATA` können entfallen, weil unser Programm kein Datensegment benötigt. Der Rahmen der Funktion lautet aufgrund der Vorbedingungen wie folgt:

```
PUBLIC  _rol

_TEXT   segment byte public 'CODE'
        assume  cs:_TEXT
```

```
_rol    proc    near
; Code
_rol    endp

_TEXT   ends
end
```

Zweckmäßigerweise erleichtert man sich den Zugriff auf die Funktionsparameter durch die Vereinbarung einer Datenstruktur, nicht unähnlich den `struct`s in "C". Nach dem Sichern des `BP`-Registers auf den Stack und der Übernahme von `SP` nach `BP` zeigt `SS:BP` auf `old_BP`. Der Parameter `n` ist, obwohl nur ein Byte breit, als Wort deklariert, weil der `push`-Befehl stets Worte auf dem Stack ablegt.

```
param STRUC
; oberstes Stapelelement
  old_BP        DW ?                    ; old_BP = 0
  old_ip        DW ?                    ; old_ip = 2
  adr_x         DW ?                    ; adr_x  = 4
  n             DW ?                    ; n      = 6
; Stapel vor Aufruf, Wachstumsrichtung ^^^
param ENDS
```

Und so sieht das Programm aus, das an der Stelle "Code" des Rahmens einzusetzen ist. Die Umdeklaration von `[BP].n` als `BYTE PTR` ist erforderlich, weil der Assembler wegen der technisch bedingten Deklaration von `n` als Wort einen Fehler melden würde.

```
  push  bp
  mov   bp, sp

  mov   bx, [bp].adr_x                  ; BX = &x
  mov   ax, [bx]                        ; AX = *BX = x
  mov   cl, BYTE PTR [bp].n             ; CL = n
  rol   ax, cl                          ; AX = rol (AX, CL)

  pop bp
  ret
```

Das Modul `rol.asm` wird mit dem Aufruf "`tasm rol`" des Turbo-Assemblers in `rol.obj` übersetzt. Für die Kompilierung des in "C" geschriebenen Testprogramms `test_rol.c` ist "`tcc -ms -c -I<Pfad zu #include-Dateien> test_rol`" einzugeben (Speichermodell `SMALL`, nur kompilieren).

```
/* Datei "TEST_ROL.C" */
extern WORD rol (WORD *x, BYTE n);

void main (void)
{ WORD x;

  x = 0xAAAA;
  printf ("rol (0xAAAA, 0x03) = %04.4X\n", rol (&x, 0x03));
};
```

Krönender Abschluß des Ganzen ist der Linkvorgang, dessen Resultat das ausführbare Programm ist. Die Argumente für den Turbo-Linker lauten, in der Reihenfolge Objektdateien, Zieldatei, Protokolldatei, Bibliotheken (ggf. Dateinamen um Verzeichnis erweitern): "`tlink cs test_rol rol, test_rol,, cs`".

3.3 Der Aufbau von MS-DOS

Das Disk Operation System der Firma Microsoft (MS-DOS) ist das Betriebssystem der meisten IBM-kompatiblen PCs, die alle auf der 8086-Familie von INTEL aufbauen. Wie kam es dazu? In den 70er Jahren war CP/M-80 von Digital Research ein weit verbreitetes Betriebssystem für Microcomputer auf Basis des 8080 (INTEL) und Z80 (Zilog). Diese Prozessoren verfügten über eine 8 Bit breite Architektur, und mit dem Aufkommen des 8086 (INTEL) entstand Bedarf an einem 16 Bit-Betriebssystem. Mitte der 70er Jahre entwickelte der Hersteller Seattle Computer Products "86-DOS", das CP/M in vielerlei Hinsicht sehr ähnlich war, um die Portierung bestehender Software zu erleichtern.

86-DOS war allerdings an Produkte von Seattle Computer Products gebunden, und so warteten andere Hersteller auf CP/M-86 von Digital Research. Unter anderem benötigte IBM für seine neue Modellreihe, die PCs, ein 16 Bit-Betriebssystem. Die neben anderen Softwarehäusern beauftragte Firma Microsoft erwarb die Rechte an 86-DOS, führte grundlegende Änderungen durch und nannte das Produkt MS-DOS, das 1981 auf den Markt kam. IBM stattete seine PCs mit diesem als PC-DOS bezeichneten Betriebssystem aus, das sich schnell gegen CP/M-86 durchsetzte. Im Laufe der Jahre wurde MS-DOS immer wieder verbessert und der Hardwareentwicklung angepaßt [8].

1987 brachte Microsoft OS/2 heraus, das endlich der Lage ist, die Fähigkeiten der INTEL-Prozessoren ab dem 80286 zu nutzen. Leider konnte sich OS/2 bis heute (1991) nicht gegen das veraltete MS-DOS durchsetzen, sondern wurde im Gegenteil durch Microsoft Windows eher noch behindert. Grund dafür ist sicherlich die weltweite Verbreitung von MS-DOS und der Unwille, sich von einem riesigen Angebot an Software zu trennen bzw. sich auf ein neues Betriebssystem einzulassen. Microsoft zeigte 1991 durch die Freigabe der Version 5.0, daß MS-DOS wohl noch lange nicht zum alten Eisen gehört. Es sollte sich aber jeder fragen, ob es sinnvoll ist, mit dem 80386 oder höher bestückte Computer in einem Modus zu betreiben, der diese sehr leistungsfähigen 32 Bit-Prozessoren zu einem schnellen 8086er degradiert.

Nach dieser kleinen Historie ist nun die Systemarchitektur von MS-DOS Gegenstand unserer Betrachtungen. Die folgenden Abschnitte erläutern den Aufbau und die Funktion von MS-DOS, soweit es für unsere Zwecke relevant ist. Der Abschnitt 4.1 "Anforderungsanalyse" im nächsten Kapitel betrachtet MS-DOS aus dem Blickwinkel der Computerviren und der Programme, die sie abwehren sollen. Zur Vertiefung seien [14] und [8] empfohlen.

3.3.1 Kommandoebene

Die Kommando- oder Programmebene ist die Ebene, die nach dem Einschalten des Computers und Laden des Betriebssystems automatisch zur Verfügung steht und mit der der reine Anwender ausschließlich zu tun hat (Abb. 3.8). Eingaben des Benutzers wie z.B. `dir` zum Auflisten des Inhaltsverzeichnisses oder `copy` zum Kopieren von Dateien werden entgegengenommen und ausgeführt, Programme und Stapeldateien können durch Eingabe des Namens gestartet werden.

Abbildung 3.8: Der Aufbau von MS-DOS

Genaugenommen wird dieser Dialog mit dem Anwender nicht durch einen Teil des Betriebssystems MS-DOS realisiert, sondern durch das Systemprogramm `command.com`, das beim Systemstart automatisch ausgeführt wird. Ein solches Programm zur Ausführung elementarer Systemoperationen nennt man *Befehls-* oder *Kommandointerpreter* (engl. shell) . Der Kommandointerpreter kommuniziert mit dem Benutzer, nimmt Eingaben entgegen, interpretiert diese und gibt die Ergebnisse der angeforderten Operationen aus. Da die Shell nur ein gewöhnliches Programm ist, kann sie vom Anwender ausgetauscht werden, z.B. gegen eine Version mit Kontrollfunktionen. `4dos` ist eine Shell für MS-DOS, die die Fähigkeiten von `command` umfaßt und stark erweitert. Unter UNIX haben z.B. die Bourne-Shell und die C-Shell breite Akzeptanz gefunden.

Interne/externe Kommandos. Die auf Kommandoebene angebotenen Befehle lassen sich in zwei Gruppen unterscheiden, deren Eigenschaften und Bedeutung für die Abwehr noch unter 4.1 “Anforderungsanalyse” zu untersuchen sind. *Interne Kommandos* sind im Kommandointerpreter selbst implementiert und werden durch interne Funktionen (→ Name) behandelt. *Externe Kommandos* sind alle Befehle, die der Befehlsinterpreter nicht selbst bearbeitet. Wird vom Benutzer ein Befehl eingegeben, der dem Kommandointerpreter unbekannt ist, versucht dieser, ein Programm mit diesem Namen zu starten. Durch den Ladevorgang ist die Ausführung externer Befehle wie `xcopy` (genauer: `xcopy.exe`) gegenüber den internen Kommandos etwas verzögert.

Fallbeispiel “type”
Ein kleines Fallbeispiel soll die Zusammenarbeit der vier Ebenen von MS-DOS begleitend zu den folgenden Abschnitten erläutern. Die gestellte Aufgabe sei die Ausgabe einer Datei auf dem Bildschirm mit Hilfe des internen Kommandos `type`. `command.com` nimmt die Eingabe

```
C:\> TYPE README.DOC
```

des Benutzers entgegen und führt eine Analyse des Textes durch (sog. Parsing). In unserem Beispiel wird der Befehl `TYPE` erkannt und die interne Funktion `type` mit dem Dateinamen `readme.doc` als Argument aufgerufen. Der Pseudocode könnte wie folgt aussehen:

```
Funktion TYPE (Dateiname)
  Datei oeffnen. Fehler (Datei nicht vorhanden)?
  Ja:
    Meldung ausgeben
  Nein:
    Datei bis zum Ende lesen und auf Bildschirm ausgeben
    Datei schliessen
Funktions-Ende
```

Und etwas konkreter in "C":

```
int type (char filename[])
{ FILE *in;                              /* Eingabefile            */
  char c;                                /* Zeichenpuffer          */

  if ((in = fopen (filename, "r")) == NULL)
  { fprintf (stderr, "file not found\n");
    return (-1);
  };
  while ((c = fgetc (in)) != EOF)
  { fputc (c, stdout);
  };
  fclose (in);
  return (0);
};
```

`type` verwendet die "C"-Funktion `fopen` zum Öffnen der Datei, `fgetc`, um aus der Datei ein Zeichen zu lesen, und `fclose`, um die Datei zu schließen. Der Begriff "Datei" ist sehr abstrakt und sagt nichts über Gerät, Speichermedium und Zugriffsverfahren aus. Die Eingabe kann z.B. von Diskette, Harddisk oder, über reservierte Dateinamen, von der seriellen Schnittstelle (`com`) oder Tastatur (`con`) erfolgen.

3.3.2 DOS-Kernel (Interruptebene I)

Das Kernel ist die höchste Abstraktionsebene unter MS-DOS. Es realisiert Dateifunktionen (auf blockorientierten Geräten), die Ein-/Ausgabe mit seriellen Geräten, die interne Speicherverwaltung sowie das Laden und Starten von Programmen. Dienste werden über Softwareinterrupts und Parameterübergabe via Register aufgerufen. Das Kernel greift selbst wieder auf primitivere Teile des Betriebssystems zurück. Dazu zählen die Gerätetreiber für die Arbeit mit seriellen und blockorientierten Geräten und Funktionen des BIOS. Durch das Kernel wird der Benutzer von den Niederungen der Hardware getrennt und muß sich beispielsweise nicht darum kümmern, ob er Daten auf Diskette, Harddisk, Drucker oder über ein Netzwerk ausgibt. Er kann jedesmal die gleichen Funktionsaufrufe verwenden, die Umsetzung und Anpassung nimmt das Betriebssystem vor.

Zur Realisierung der Dateizugriffe setzt das Kernel die logische Dateistruktur in eine logische Sektorstruktur um, die vom Sektor null bis zur höchsten verfügbaren

Sektornummer des Datenträgers reicht. Zum Bearbeiten der Datei- oder von Dateiverwaltungsdaten erteilt das Kernel dem Gerätetreiber eine Folge von Aufträgen, ab einem bestimmten Startsektor eine Anzahl von Blöcken in einen Puffer zu lesen oder aus dem Puffer auf das Speichermedium zu schreiben.

Fallbeispiel "type"
Verfolgen wir die Umsetzung der "C"-Funktion `fgetc` durch den Compiler in Aufrufe des Betriebssystems. Über den DOS-Funktions-Interrupt 21_{16} werden alle Funktionen zur Dateibearbeitung aufgerufen. `fgetc` benutzt die Funktion $3F_{16}$ "Read File or Device", die ein Zeichen aus einer geöffneten Datei liest. In diesem Fall kommt das Kernel dem Compiler sehr entgegen. Von der internen Dateiverwaltung der "C"-Bibliothek einmal abgesehen, entspricht der "C" Funktionsaufruf direkt einem Interrupt-Aufruf.

Intern läuft sehr viel mehr ab, denn wir wollen sequentiell lesen, obwohl Disketten zu den blockorientierten Speichern gehören und der Datentransfer stets in Paketen zu 512 Bytes erfolgt. Das Kernel nimmt deshalb eine Umsetzung vor und sorgt dafür, daß, wenn der gerade bearbeitete Block zu Ende ist, der nächste Block der Datei vom Gerätetreiber eingeladen wird. Die Blockfolge und die Lage der Blöcke werden aus dem FAT (s.a. 3.5) ermittelt. Das Erreichen des Dateiendes oder Auftreten eines Fehlers wird dem aufrufenden Programm durch Setzen des Carry-Flags und Übermittlung einer Fehlernummer im `AX`-Register signalisiert.

3.3.3 Gerätetreiber

Gerätetreiber fungieren als Mittler zwischen Kernel und BIOS. Das Kernel arbeitet geräteneutral, d.h. es sieht nicht die tatsächliche Datenorganisation und Funktion der Peripherie, sondern nur standardisierte Datenformate und Transportfunktionen. Unabhängig davon, ob das angeschlossene Gerät eine Festplatte, ein Magnetband oder eine RAM-Disk ist, vermitteln die Gerätetreiber eine einheitliche Schnittstelle. MS-DOS, wie z.B. auch UNIX, unterscheidet zwei elementare Geräteklassen — die blockorientierten und die seriellen Geräte.

Block Device Driver. Peripheriegeräte, die an einen Computer angeschlossen werden können, verhalten sich im Detail meist unterschiedlich, sind sich aber im Prinzip doch recht ähnlich. Als Beispiel hierfür seien Festplatten genannt, die sich je nach verwendetem Controller (Seagate-Interface, AT-Bus, SCSI, ESDI etc.), Anzahl der Zylinder, Köpfe und Sektoren stark voneinander unterscheiden. Dennoch ist allen die Aufteilung der Daten in Speicherblöcke gemeinsam, die als kleinste Transporteinheit im wahlfreien Zugriff, d.h. über die Angabe einer Adresse, gelesen oder geschrieben werden können.

Anwenderprogramme können blockorientierte Gerätetreiber nicht direkt ansprechen, sondern nur indirekt über das Kernel. Dieses wählt bei Dateioperationen automatisch den Gerätetreiber aus, der für das Laufwerk, auf dem sich die Datei befindet, zuständig ist. Der Gerätetreiber konvertiert logische Sektoren in reale Sektoren auf dem Datenträger. Aus einer linearen Adresse, der logischen Sektornummer, werden Positionsangaben wie Zylinder, Kopf und Sektor. Der ROM-BIOS-interne Treiber für zwei Disketten- und zwei Festplattenlaufwerke benutzt BIOS-Funktionen, um einzelne Sektoren zu lesen.

Andere Gerätetreiber für z.B. RAM-Disks und Magnetbänder verwenden eigene Routinen, um die Organisation umzusetzen und auf die Hardware zuzugreifen. Für Laufwerke, die nicht über einen Standard-Controller angeschlossen sind, ist ein entsprechender Treiber durch eine "`device=`"-Anweisung in der Datei `config.sys` einzubinden. Zusätzliche Blockgerätetreiber werden immer nach den DOS-Treibern installiert und tragen daher Laufwerksbuchstaben, die hinter denen des BIOS-Treibers liegen.

Character Device Driver. Die zweite Geräteklasse ist die der zeichenorientierten Geräte, zu denen z.B. der Drucker (parallele Schnittstelle), die Maus (serielle Schnittstelle), die Zeichenausgabe auf den Bildschirm und das Lesen von Zeichen von der Tastatur gehören. Die kleinste Transporteinheit ist hier das einzelne Zeichen.

Zeichenorientierte Geräte werden vom Kernel — ähnlich wie bei UNIX — als Dateien verwaltet, aus denen gelesen und auf die geschrieben werden kann (Tab. 3.7). Das hat den Vorteil, daß sich die Ein- und Ausgabe von und auf bestimmte Geräte leicht auf andere Kanäle umlenken läßt ("I/O Redirection"). Der Kommandointerpreter gestattet beispielsweise, bei der Absendung eines Befehls Dateien für die Ein-/Ausgabe anzugeben. Programme, die von der Standardeingabe (Tastatur, `stdin`) lesen und in die Standardausgabe (Bildschirm, `stdout`) schreiben, lesen und schreiben dann in die angegeben Dateien.

Dateiname	Bezeichnung
Character Devices	
CON	Konsole (Eingabe von Tastatur; Ausgabe auf Bildschirm)
AUX	serielle Schnittstelle (synonym zu COM1)
COM1-4	serielle Schnittstelle
PRT	parallele Schnittstelle (synonym zu LPT1)
LPT1-3	parallele Schnittstelle
Block Devices	
`A:`, `B:`, ...	Disketten- und Festplattenlaufwerke

Tabelle 3.7: Gerätenamen

Beispielsweise liest das MS-DOS-Programm `sort` normalerweise von einer Datei namens `stdin`, die mit der Tastatur in Verbindung steht, und nimmt solange durch `Return` getrennte Daten an, bis das Dateiendezeichen `Control` `Z` erkannt wird. Nach dem Sortiervorgang gibt das Programm die Information auf die Datei `stdout` aus, welche die Daten an den Bildschirm weitergibt. Dahinter steckt sowohl bei `stdin` als auch bei `stdout` die Datei oder genauer der Gerätetreiber `con`, der fester Bestandteil des BIOS ist. Die Ein- oder Ausgabe auf oder von der Datei `con` hätte genau den selben Effekt.

Zeichentreiber werden, außer dem "`nul`"-Treiber, an der Spitze der Treiberliste eingefügt. Das Kernel durchsucht beim Zugriff auf ein Gerät die Liste der Gerätetreiber von vorne nach hinten und stoppt beim ersten passenden Eintrag. Dies bedeutet, daß später geladenen Zeichentreiber ältere gleichen Namens ersetzen. Ein Beispiel dafür ist `ansi.sys`, der den Treiber mit dem Namen `con` ersetzt.

Anmerkungen. Bei Dateioperationen mit Gerätedateien gilt es einige Dinge zu beachten, die beim Umgang mit normalen Dateien keine Rolle spielen. Im Gegensatz zu konventionellen Dateien unterscheidet das Kernel bei diesem Typ einen ASCII- und einen Binär- oder Binary-Modus, an dessen Realisierung der Gerätetreiber nicht beteiligt ist. Im ASCII-Modus transportiert das Kernel immer nur ein Zeichen auf einmal zwischen dem Gerät und einem internen Puffer. Die Eingabe stoppt, falls ein Dateiendezeichen EOF ($1A_{16}$; Control Z) erkannt wird. Im Falle von CR (13_{16}; Return) gibt das Kernel den Inhalt seines internen Puffers weiter.

Im Binär-Modus wird exakt die bestellte Anzahl von Zeichen eingelesen oder geschrieben. Da die Daten im Eingabestrom nicht interpretiert werden, bleiben Steuerzeichen wie Dateiende und CR wirkungslos. Eingabe von Tastatur wird somit unmöglich, es sei denn, es werden immer nur ein Zeichen auf einmal angefordert und entsprechende Reaktionen selbst implementiert. Der Binär-Modus ist besonders dann fatal, wenn das aufrufende Programm z.B. von der seriellen Schnittstelle mehr Zeichen zum Lesen angefordert hat, als der Sender schickt. MS-DOS wartet dann beliebig lange auf ein Zeichen, ohne jemals zum Auftraggeber zurückzukehren. Nur ein Reset hat dann noch die nötige Überzeugungskraft.

Interna. Die Kommunikation Kernel/Gerätetreiber erfolgt nicht über Softwareinterrupts, sondern über direkte Aufrufe des Treibers. Die Tabelle der Einsprungpunkte wird bei der Initialisierung der Treiber im Urladeprozeß erstellt. Diese Adreßtabelle ist im Gegensatz zur Vektortabelle der Interrupts nicht öffentlich zugänglich oder über Kernelfunktionen manipulierbar. Die Parameterübergabe läuft über Prozessorregister und spezielle Nachrichtenblöcke.

Das Gesagte impliziert, daß Aufrufe der Gerätetreiber durch das Kernel nicht kontrollierbar sind. Falls die Treiber keine BIOS-Funktionen verwenden, kann eine Kontrolle allenfalls auf Kernelebene erfolgen. So führt z.B. ein Zugriff auf RAM-Disk zu keinen Reaktionen auf BIOS-Ebene. Der zuständige Gerätetreiber nimmt neben Anpassung der Organisation (lineare RAM-Adressen in Blöcke) auch gleich den Zugriff (Lesen und Schreiben von Speicherinhalten) vor. Ähnliches gilt für Treiber, die direkt mit einem Controller kommunizieren und daher das BIOS weder brauchen können noch benutzen.

Fallbeispiel “type”
Der Gerätetreiber erhält vom Kernel Anweisungen zum Lesen logischer Sektoren. Dieser rechnet mit seinem Wissen über die Organisation des Datenträgers die lineare Sektoradresse in eine Positionsangabe mit Zylinder, Kopf und Sektor um. Im Fall des im ROM-BIOS eingebauten Treibers erfolgt der eigentliche Zugriff über Funktionen des BIOS.

3.3.4 BIOS (Interruptebene II)

Das BIOS (**B**asic **I**nput **O**utput **S**ystem = Grundlegendes Ein-/Ausgabe-System) stellt die Schnittstelle zwischen Hard- und Software dar, zu der Anwenderprogramme, das Kernel sowie die Gerätetreiber gehören. Es besteht aus zwei Teilen, dem ROM-BIOS, das sofort nach dem Einschalten benutzt werden kann, und dem RAM-BIOS, das im Laufe des Bootprozesses geladen wird. Der ROM-Teil stellt zusammen mit eingebauten

Gerätetreibern elementare Funktionen zur Ein- und Ausgabe bereit, wie sie u.a. zum Laden des Betriebssystems von Diskette oder Festplatte notwendig sind.

Alle Aufrufe des BIOS erfolgen über Interrupts, die durch Software oder, ein neuer Aspekt, Hardwarebausteine wie z.B. den Tastaturcontroller ausgelöst werden. Anwenderprogramme rufen Dienste wie beim Kernel über verschiedene Interruptnummern auf, die ihrerseits evtl. noch weiter in Funktionen und Subfunktionen aufgegliedert sind. Eine Übersicht über die Gesamtheit der Interrupts vermittelt die Tabelle C.3 im Anhang C.1. Die Besprechung der einzelnen Funktionen würde leicht ein eigenes Buch füllen und erfolgt daher nur in dem Maße, wie es für unsere Zwecke notwendig ist.

Hardware-Service. Die Interruptnummern 00_{16} bis $0F_{16}$ sind der Hardware vorbehalten und noch einmal in zwei Gruppen unterteilt. In der ersten, von 00_{16} bis 07_{16}, finden sich u.a. Unterbrechungen, die die CPU bei bestimmten Ereignissen intern auslöst. Dazu zählen die Division durch Null, die Abarbeitung eines Befehls im Einzelschrittmodus (Trace oder Single Step), die Abarbeitung eines unzulässigen Befehls und andere. Die zweite Gruppe von 08_{16} bis $0F_{16}$ beinhaltet die extern ausgelösten Unterbrechungen durch den Systemtimer, den Tastaturcontroller, die Bausteine für die seriellen und parallelen Schnittstellen sowie den Disk-Controller.

Software-Service. Die 16 Interrupts von 10_{16} bis $1F_{16}$ umfassen die für Anwender und Betriebssystem aufrufbaren Funktionen. Ein Blick in die Tabelle zeigt, daß es sich bei den angebotenen Diensten überwiegend um elementare Ein- und Ausgabe von und auf Bildschirm, Diskette/Festplatte, serielle/parallele Schnittstelle, Tastatur und Drucker handelt. Das BIOS hat in diesem Zusammenhang zwei Aufgaben. Zum einen nimmt es dem Benutzer die Initialisierung und den Dialog mit den Hardwarebausteinen ab und hebt die Schnittstelle schon recht komfortabel auf Registerebene an. Es setzt Anforderungen in Kommandos für den spezifischen Controller um und empfängt von diesem Daten, Bestätigungen und Fehlermeldungen.

Zum anderen nimmt es die Anpassung zwischen unterschiedlichen Hardwarekomponenten vor. Dies gilt besonders für die Vielfalt der Grafikkarten, die z.T. ihr eigenes BIOS in einem ROM mitbringen, das beim Bootvorgang eingebunden wird und den Video-Interrupt 10_{16} übernimmt. Diese Methode verwenden auch einige Festplattencontroller, die den Disk-Interrupt 13_{16} kontrollieren. Über das Abfangen von Interrupts lassen sich also sogar Funktionen des ROM-BIOS verändern und erweitern.

3.4 Verwaltung interner Speicher

Unter internem Speicher werden alle Daten speichernde Hardwarekomponenten verstanden, die der Prozessor direkt über Adreß- und Datenbus ansprechen kann. Dazu zählen Schreib-/Lese-Speicher wie RAMs und EEPROMs und Nur-Lese-Speicher wie ROMs, PROMs und EPROMs. Manche davon sind flüchtig, verlieren also beim Abschalten der Stromversorgung ihren Inhalt (RAMs), manche sind nur einmal beschreibbar (PROMs) und wieder andere lassen sich durch Bestrahlung des Chips mit UV-Licht wieder löschen (EPROMs).

3.4.1 Organisation des Arbeitsspeichers

Bei den Festwertspeichern gibt es nichts dynamisch zu verwalten, darum wenden wir uns gleich den flüchtigen Speichern zu. Das RAM dient als Arbeitsspeicher für Programme und Daten und muß vom Betriebssystem geeignet organisiert und verwaltet werden. MS-DOS teilt den Speicher in Blöcke auf, die als vorwärts verkettete sequentielle Liste organisiert sind (Abb. 3.9).

Eine solche Verwaltungseinheit besteht aus einem sog. Speicher-Kontroll-Block oder MCB (von engl. **m**emory **c**ontrol **b**lock) und dem dazugehörigen, unmittelbar folgenden reservierten Speicher. Der MCB ist 16 Bytes oder ein Paragraph lang und wie in Tab. 3.8 beschrieben aufgebaut.

Offset	Typ	Bedeutung
00_{16}	Byte	'M': MCB folgt; 'Z': letzter MCB
01_{16}	Word	Segmentadresse des zugehörigen PSP
03_{16}	Word	Größe des Blocks in Paragraphs
05_{16}	···	
⋮	⋮	Name des Besitzers (undok.)
$0F_{16}$	···	

Tabelle 3.8: Aufbau MCB (Memory Control Block)

Weil der MCB selbst ein Paragraph groß ist, beginnt der nächste Speicherblock an der $\text{Segmentadresse}_{nextMCB} = \text{Segmentadresse}_{MCB} + \text{Blockgröße} + 1$. Das Programm, zu dem der Speicherblock gehört, kann über den Zeiger auf den PSP ermittelt werden, dessen Bedeutung der nächste Abschnitt erläutert. Will man die Liste der MCBs durchforsten, stößt man auf ein Problem: Wo befindet sich der erste MCB? Es existiert offiziell keine Betriebssystemfunktion, die darüber Auskunft geben könnte. Die nicht dokumentierte Funktion 52_{16} "Get List of Lists" des DOS-Funktionsinterrupts schafft Abhilfe. Sie liefert in `ES:BX` einen FAR-Zeiger auf eine Datenstruktur, die selbst wieder Zeiger enthält, darunter auch einen auf den Beginn (Wurzel, Anker) der MCB-Kette (Tab. 3.9).

Besonders hingewiesen sei auf die Tatsache, daß der Zeiger in `ES:BX` nicht auf den Start der Tabelle verweist, sondern erst um vier Bytes dekrementiert werden muß (daher auch der negative Offset des ersten Eintrags). Unsere "C"-Funktion `get_lol` nimmt diese Korrektur automatisch vor, so daß der erste Eintrag an Offset 0 steht.

Beispiel "ReadMCB"
Betrachten wir das Beispielprogramm `ReadMCB`, das die Kette der MCBs durchgeht und dabei Adresse, Größe und Besitzer der Speicherblöcke anzeigt. Ein solches Programm ist bei der Suche nach Softwareanomalien sehr nützlich, die auf konventionelle Weise Speicher reservieren. Tauchen in der Liste Programme auf, die nicht vom Anwender installiert wurden oder keinen Namen tragen? Sind legale TSR-Programme größer als üblich? Ist der freie Speicher kleiner als erwartet? Existieren Speicherblöcke, die scheinbar keinem Programm zugeordnet sind? All diese Fakten weisen auf speicherresidente

Abbildung 3.9: Speicherverwaltung unter MS-DOS

Viren hin. Durch seine praktische Anwendbarkeit ist `ReadMCB` das erste Programm unseres Antivirus-Pakets.

Für `ReadMCB` benötigen wir einige Definitionen, Makros und Typdefinitionen. `BYTE`, `WORD` und `DWORD` (Double Word) sind Synonyme für vorzeichenlose Variablen mit den Längen 1, 2 und 4 Bytes; die anderen Bezeichner sind symbolische Konstanten für boolesche Ausdrücke.

```
#define BYTE  unsigned char
#define WORD  unsigned short
#define DWORD unsigned long
#define FALSE           0
#define TRUE            1
#define ON              0
#define OFF             1
#define NO              0
#define YES             1
```

Mit dem Makro `MAKE_FARPTR` kann ein FAR-Zeiger erzeugt, mit den Makros `GET_SEG` und `GET_OFF` in Segment und Offset zerlegt werden. `PARA` schließlich bestimmt von einer FAR-Adresse die nächst tieferliegende Segmentadresse.

Offset	Typ	Bedeutung
-04_{16}	FAR*	erster MCB
00_{16}	FAR*	erster DPB (**D**isk **P**arameter **B**lock)
04_{16}	FAR*	SFT (**S**ystem **F**ile **T**able)
08_{16}	FAR*	Clock-Device (Uhr)
$0C_{16}$	FAR*	Console-Device (Tastatur, Bildschirm)
10_{16}	word	max. Sektorlänge (512)
12_{16}	FAR*	erster DOS-Diskpuffer
16_{16}	FAR*	interne Daten zu log. Laufwerken (s.a. Anhang C.2)
$1A_{16}$	···	unbekannt
⋮	⋮	⋮

Tabelle 3.9: Aufbau "List of Lists"

```
#define MAKE_FARPTR(seg, off) \
                    ((void far *) (((DWORD)(seg) << 16) | (WORD)(off)))
#define GET_OFF(fp) ((WORD)((DWORD)(fp) & 0xFFFF))
#define GET_SEG(fp) ((WORD)((DWORD)(fp) >> 16))
#define PARA(fp)    ((WORD)(((DWORD)(fp) >> 16) + \
                    (((DWORD)(fp) & 0xFFFF) >> 4)))
```

Die Strukturtypen `T_LOL` und `T_MCB` entsprechen den internen MS-DOS-Strukturen für die "List of Lists" und MCBs.

```
struct T_LOL                            /* "List of Lists"         */
{ struct T_MCB far  *mcb_anchor;        /* Zeiger auf ersten MCB   */
  void far          *dpb_anchor;        /* Zeiger auf ersten DPB   */
  struct T_SFT far  *sft;               /* Zeiger auf SFT          */
  void far          *clock_dev;         /* Zeiger auf CLOCK-Device */
  void far          *con_dev;           /* Zeiger auf CON-Device   */
  WORD              max_sec_len;        /* max. Sektorgroesse (512)*/
  void far          *buffer_q;          /* Adresse DOS Diskpuffer  */
  void far          *dev_buffer;        /* int. Daten log. Laufw.  */
};

struct T_MCB                            /* Memory Control Block    */
{ char  flag;                           /* M:o.k.; Z:letzter Block */
  WORD  owner;                          /* Segmentadresse PSP      */
  WORD  paras;                          /* Groesse in Paragraphs   */
  BYTE  res[3];                         /* reserviert              */
  BYTE  name[8];                        /* [Programmname, undok.]  */
};
```

Mit diesem Vorwissen ausgestattet können wir uns an den eigentlichen Programmtext wagen. Die Funktion `getlol` liefert einen FAR-Zeiger auf die "List of Lists" (Tab. A.15). Mit dieser Angabe können wir den FAR-Zeiger `mcb_ptr` auf den ersten MCB setzen. Der verketteten Liste wird nun so lange gefolgt, bis festgestellt wird, daß der zuletzt bearbeitete Block der letzte, mit 'Z' markierte, war. Die Segmentadresse des jeweils nächsten MCBs ergibt sich aus der Segmentadresse des alten MCBs plus die Größe des Speicherblocks in Paragraphs plus die Länge eines MCBs (ein Paragraph). Der Offset ist immer 0000_{16}. Die MCB-Liste ist so lange zu durchlaufen, bis der letzte Block bearbeitet *wurde* (daher der Umweg über `last`). Noch eine Anmerkung zum Li-

sting: Die weiter unten besprochene Funktion `get_mcb_name` ermittelt den Namen des Besitzers eines MCBs.

```
void main (void)
{ struct T_MCB far *mcb_ptr;            /* Zeiger auf MCB          */
  char name[65];                        /* Name des Besitzers      */
  char last;                            /* letzte MCB-Kennung      */

  mcb_ptr = getlol () -> mcb_anchor;
  printf ("Address Owner Env.   Size Name\n");
  do
  {   printf ("%04.4X    %04.4X  %04.4X %6lu %s\n",
      PARA (mcb_ptr) + 1,               /* Start des res. Speichers*/
      mcb_ptr -> owner,                 /* Besitzer des Speichers  */
      /* Segmentadresse Environment, s.a. Beschreibung des PSP     */
      *(WORD far *)MAKE_FARPTR (mcb_ptr -> owner, 0x2C),
      (DWORD)(mcb_ptr -> paras) << 4,   /* Groesse in Bytes  */
      get_mcb_name (mcb_ptr, name));    /* Name des Besitzers      */

    last = mcb_ptr -> flag;
    mcb_ptr = MAKE_FARPTR (GET_SEG (mcb_ptr) + mcb_ptr -> paras + 1,
                           GET_OFF (mcb_ptr));
  } while (last != 'Z');
};
```

Anhand der MCB-Liste und Informationen über die Speicherbelegung durch das Betriebssystem (s.a. 3.5.3 "Der Urladevorgang") läßt sich eine vollständige Belegungskarte des Adreßraums erstellen, die für die zu erstellende Sicherheitssoftware noch wesentlich sein wird.

Für Speicheroperationen stellt das Kernel eine Reihe von Funktionen zur Verfügung, von denen wir allerdings nur die Nummer 49_{16} "Release Memory Block" (Speicherblock freigeben) benötigen werden (Tab. A.12). Das hängt u.a. damit zusammen, daß bei der Speicherreservierung "C"- und Kernel-Funktionen nicht miteinander zusammenarbeiten und ein Crash bei gleichzeitiger Benutzung wahrscheinlich wäre.

3.4.2 Start und Struktur von Programmen

Programme werden aufgerufen, indem entweder der Name des Programms in der Kommandozeile von `command.com` eingegeben oder das Programm von einem anderen (z.B. als Overlay) geladen und gestartet wird. In beiden Fällen wird die Kernel-Funktion $4B_{16}$ "Load & Execute" aufgerufen (Tab. A.13). MS-DOS reserviert dazu Speicher für die Environmentvariablen und den Programmcode in zwei getrennten Blöcken. Dem Programmcode wird der Programmvorspann (engl. **p**rogram **s**egment **p**refix, kurz PSP) vorangestellt (s.a.Abb. 4.10).

Zu beachten ist, daß MS-DOS beim Laden für das Programm den gesamten freien Speicher reserviert und deshalb keine weiteren Programmstarts oder Reservierungen möglich sind. Jedes Programm sollte daher als erste Aktion seinen tatsächlichen Speicherbedarf feststellen und im eigenen Interesse und dem der nachfolgenden Programme nicht benötigten Speicher wieder freigeben (s.a. 3.7.1 "TSR-Programme"). Diese Aufgabe erledigt bei in "C" geschriebenen Programmen der automatisch eingebun-

dene Startup-Code. Da dieser etwas großzügig arbeitet, sollten speicherresidente Programme selbst ihren Platzbedarf ermitteln und dem Betriebssystem mitteilen.

Program Segment Prefix (PSP). Im PSP befinden sich Informationen über das Programm und Hilfsdaten für das Betriebssystem (Tab. 3.10). Die Segmentadresse des PSP liefert die Turbo-C-Funktion

```
unsigned getpsp (void)
```

(Tab. A.18). Der Startup-Code der Turbo-C-Programme wertet die im PSP gespeicherte Parameterzeile aus und legt Zeiger auf die einzelnen Argumente im Array `char *argv[]` (**arg**ument **v**alues) ab. `argv[0]` ist für den vollständigen Programmnamen reserviert, aus dem der Startpfad ermittelt werden kann. Die Anzahl der Argumente plus eins ist in `int argv` (**arg**ument **c**ount) festgehalten.

Environment. Zu den Daten im PSP gehört ein Zeiger auf den Speicherblock mit den Umgebungsvariablen des Programms. Dieser Block wird kürzer auch als Umgebung (engl. environment) bezeichnet. Die Environmentvariablen werden von `command.com` verwaltet und vom Betriebssystem unterstützt. Da sie global und für jedes Programm verfügbar sind, eignen sie sich für die Definition von gewissen Rahmenbedingungen, einer Programm*umgebung* (→ Name). So bestimmt z.B. `prompt` das Aussehen der Eingabeaufforderung und `path` die Suchpfade für Programme, weil `command.com` auf diese Umgebungsvariablen reagiert.

Die Struktur des Environmentblocks ist wie folgt:

```
Environment ::= {<Definition>'\0'}'\0''\1''\0'<Programmname>'\0'
Definition  ::= <Variablenname>=<Text>
```

Sowohl die Variablendefinitionen als auch der Programmname sind gemäß der "C"-Konvention mit einem `0`-Byte abgeschlossen. Das Ende der Liste der Environmentvariablen ist durch ein zusätzliches `0`-Byte markiert. Der vollständige Programmname folgt nach einem `1`- und einem weiteren `0`-Byte. Da dieser den kompletten Startpfad enthält, kann das Programm bestimmen, in welchem Verzeichnis die eigene Programmdatei und evtl. benötigte Hilfsdateien stehen (nicht zu Verwechseln mit dem aktuellen Verzeichnis!). Der Startcode von Turbo-C plaziert einen Zeiger auf den vollständigen Programmnamen in `argv[0]`.

Weil Environment und Programmcode in zwei separaten Blöcken untergebracht sind, kann das Programm den Platz für das Environment freigeben, falls es dies nicht mehr benötigt. Die meisten Programme werten die Umgebungsvariablen, Startparameter und -pfad am Anfang aus und können später auf diese Informationen verzichten. Auf diese Weise läßt sich Speicherplatz einsparen. Das Turbo-C Startmodul macht diese Erfolge teilweise wieder zunichte, weil es das Environment vor dem Aufruf von `_main` in eigens dafür reservierten Speicher kopiert. Immerhin fällt trotzdem die Doppelbelegung weg.

Beispiel "ReadEnv"
Das Übungsprogramm `ReadEnv` zeigt, wie man Environmentvariablen und den Programmnamen auslesen kann. Der Zeiger `ptr` wird auf das eigene Environment gesetzt. Anschließend werden solange Zeichenketten mit Variablendefinitionen ausgelesen, bis ein '0'-Byte das Ende der Reihe signalisiert.

Offset	Typ	Bedeutung
00_{16}	code	"`int 20h`" → Programm beenden
02_{16}	word	Segmentadresse Anfang freier Speicher
04_{16}	byte	reserviert
05_{16}	code	"`call far` <DOS-Verteiler>" (Funktionsnummer in `CL`)
$0A_{16}$	FAR*	Adresse Programmende (`INT` 22_{16})
$0E_{16}$	FAR*	Adresse `Ctrl` `Break` -Routine (`INT` 23_{16})
12_{16}	FAR*	Adresse Critical Error Handler (`INT` 24_{16})
16_{16}	word	Segmentadresse des PSPs des aufrufenden Programms (undok.)
18_{16}	···	JFT (Job File Table) der offenen Dateien mit Verweisen in den SFT (System File Table; undok.)
⋮	⋮	20 JFT-Einträge
$2C_{16}$	word	Segmentadresse Environment
$2E_{16}$	FAR*	Adresse Systemstack (undok.)
32_{16}	word	max. Anzahl offener Dateien (= Größe JFT; 14_{16} = 20)
34_{16}	FAR*	Adresse Indextabelle der offenen Dateien (relativ zu PSP; normal 0018_{16}; undok.)
38_{16}	···	reserviert
⋮	⋮	⋮
50_{16}	code	"`int` 21h, `retf`"
53_{16}	···	reserviert
⋮	⋮	⋮
$5C_{16}$	FCB	FCB 1 (bei Start: Parameter 1 als Dateiname)
$6C_{16}$	FCB	FCB 2 (bei Start: Parameter 2 als Dateiname)
80_{16}	DTA	DTA (bei Start: Länge Parameterzeile)
82_{16}	DTA	Fortsetzung DTA (bei Start: Parameter)

Tabelle 3.10: Aufbau PSP (Program Segment Prefix)

```
void main (void)
{ char far *ptr;                          /* Zeiger in environment  */

  ptr = MAKE_FARPTR
    (*(WORD far *)MAKE_FARPTR (xgetpsp (), 0x2C), 0);
  while (*ptr)                            /* bis zur doppelten '0'  */
  { while (*ptr)                          /* gebe Variable aus      */
    { putch (*(ptr++));
    };
    putch ('\r'); putch ('\n');
    ptr++;
  };
```

Die folgende '1'-'0'-Sequenz wird auf Vorhandensein überprüft und übersprungen, um den folgenden Programmnamen auszugeben.

```
  if (*(++ptr) == 1)                    /* Programmname folgt?    */
  { ptr += 2;                           /* '\1'-'\0' ueberspringen*/
    while (*ptr)                        /* gebe Programmname aus  */
    { putch (*(ptr++));
    };
  };
};
```

Funktion "get_mcb_name"

Daraus, daß man über die logische Kette PSP→Environment den Programmnamen feststellen kann, ergeben sich eine Reihe von Anwendungsmöglichkeiten. Eine einfaches Beispiel ist das bereits entwickelte Programm `ReadMCB`, das nicht nur die PSP-Adresse des Besitzers ausgibt, sondern auch dessen Namen. Diese Aufgabe realisiert die Funktion `get_mcb_name`, die als Parameter einen Zeiger auf einen MCB erwartet und den Namen des Besitzers in den übergebenen String einträgt.

Werfen wir einen Blick auf die Arbeitsweise der Funktion. Ist die Segmentadresse `owner` des Besitzers gleich 0000_{16}, ist der betreffende Speicherblock frei.

```
char *get_mcb_name (struct T_MCB far *mcb_ptr, char name[])
{ int p;
  WORD owner, env_owner;
  char far *env, eflag;

  owner = mcb_ptr -> owner;
  if (owner == 0x0000)                  /* freier Speicher?       */
  { strcpy (name, "<free memory>");
  }
  else                                  /* Programm               */
```

Um einiges komplexer wird die Sachlage bei einem belegten MCB. Zunächst wird überprüft, ob der Eigentümer des MCBs überhaupt ein Environment besitzt, aus dem sich der Programmname bestimmen läßt. `env` ist der über den PSP des Besitzers ermittelte FAR-Zeiger auf das vorgebliche Environment.

"Vorgeblich" deshalb, weil ein Programm sein Environment freigeben kann, ohne daß der Verweis darauf im PSP automatisch gelöscht wird. Resultat ist ein *verwaister* Environment-Zeiger, der nicht benutzt werden darf. Der Grund: Der freigegebene Block kann frei bleiben oder aber vom nächsten gestarteten Programm benutzt und überschrieben werden (Abb. 3.10). Resultat ist, daß der Name des Programms nicht mehr sicher ermittelt werden kann. Der Zeiger auf das Environment `env` ist demnach nur gültig, wenn der Speicherblock, auf den er verweist, dem betrachteten Programm gehört.

Als nächster Schritt wird das `eflag` (**E**nvironment Flag) gesetzt, falls der momentan betrachtete MCB den Speicherblock kontrolliert, der das Environment seines Besitzers enthält. Diese Prüfung muß jetzt erfolgen, weil der Aufruf von `get_owner_mcb` in der nächsten Zeile den Zeiger `mcb_ptr` verändert.

```
  { env = MAKE_FARPTR (*(WORD far *)MAKE_FARPTR (owner, 0x2C), 0);
    /* EFLAG = (dieser Block ist Environment des Besitzers)       */
    eflag = (PARA (env) == PARA (mcb_ptr) + 1);
```

Mit Hilfe der weiter unten besprochenen Funktion `get_owner_mcb` wird nun der tatsächliche Besitzer `env_owner` des Environments bestimmt. Das Environment ist

Abbildung 3.10: Verwaister Environment-Zeiger

ungültig, falls der tatsächliche `env_owner` und der vorgebliche Besitzer `owner` des Environments nicht identisch sind oder das Environment nicht am Anfang des Speicherblocks beginnt. Der Besitzer des MCBs ist dann entweder ein Teil des Betriebssystems oder kann schlicht nicht ermittelt werden.

```
/* Kein Environment (angegebenes Environment gehoert nicht dem
   Besitzer oder beginnt nicht am Anfang des Speicherblocks)?  */
env_owner = get_owner_mcb (env, &mcb_ptr);
if ((owner != env_owner) || (PARA (env) != PARA (mcb_ptr) + 1))
{ if (owner <= 0x0050)              /* RAM-BIOS?               */
  { strcpy (name, "<RAM-BIOS>");
  }
  else
  { strcpy (name, "<no environment>");
  };
}
else                                /* Environment gueltig     */
```

Falls das Environment gültig ist, kann nach den langen Präliminarien endlich der Name ermittelt werden. Analog zum Verfahren in `ReadEnv` werden die Umgebungsvariablen überlesen. Falls der Programmname folgt, wird der Name in `name` kopiert, andernfalls angenommen, daß es sich um den Kommandointerpreter handelt[4]. Enthält

[4] `command.com` benutzt ein spezielles Environment.

der untersuchte MCB das Environment seines Besitzers (`eflag` gesetzt), wird der Name entsprechend markiert.

```
      { do                                  /* suchen Programmname     */
        { while (*(env++)) {};              /* Vorspulen zu Stringende */
        } while (*env);                     /* Bis Ende Variablen      */
        if (*(++env))                       /* Programmname folgt?     */
        { env += 2;
          p    = 0;
          while ((name[p++] = *(env++)) != '\0') {};
        }
        else                                /* Kein Programmname       */
        { strcpy (name, "COMMAND.COM");
        };
      };
      if (eflag)
      { strcat (name, " (environment)");
      };
    };
    return (name);
  };
```

Funktion "Get MCB Name":
`char *get_mcb_name (struct T_MCB far *mcb_ptr, char name[])`

Aufrufparameter:

`mcb_ptr`	FAR-Zeiger auf MCB
`name`	Pufferadresse (mind. 64 Bytes)

Seiteneffekte:

`name`	Name des Besitzers

Rückgabewert:
Zeiger auf `name`

Tabelle 3.11: `get_mcb_name`: Ermittle Name des Besitzers eines MCBs

Funktion "get_owner_mcb"
Über die oben erwähnte Speicherbelegungskarte (Memory Map) kann ermittelt werden, welche Speicheradresse zu welchem Programm gehört. Das ist für viele Belange der Systemsicherheit nützlich. So kann z.B. festgestellt werden, welches Programm welchen Interruptvektor bedient. Die Hauptanwendung von `get_owner_mcb` wird bei uns darin bestehen, den Auslöser eines Softwareinterrupts zu ermitteln. Wie gezeigt, ist es nicht besonders schwer, einen Interrupt abzufangen und die Parameter zu überprüfen. Woher aber weiß das Kontrollprogramm, welches Programm den Interrupt aufgerufen hat?

Bei einem Hard- oder Softwareinterrupt legt der Prozessor die Adresse, bei der das laufende Programm unterbrochen wird, und den Zustand des Flagregisters auf dem Stack ab. Ein Programm, das sich in diesen Interrupt eingeklinkt hat, kann die Rücksprungadresse und das Statuswort wie normale Funktionsparameter lesen. Weil der Ort der Unterbrechung im aufrufenden Programm liegen muß, läßt sich über die Rücksprun-

gadresse, zusammen mit der Belegungskarte des Speichers, das aufrufende Programm bestimmen. Das Kontrollprogramm kann nun unter Berücksichtigung der Aufrufparameter entscheiden, ob das Programm die angeforderte Operation durchführen darf oder nicht.

Die Funktion `get_owner_mcb` ermittelt für eine angegebene FAR-Adresse den MCB, in dessen Speicherbereich sie liegt. Dazu durchläuft eine `while`-Schleife analog zu `ReadMCB` die Liste der MCBs. Liegt die gesuchte Adresse `adr` im gerade bearbeiteten Block, wird der Wert von `mcb_ptr` in `tmp` zwischengespeichert, da sich dieser vor Verlassen der Schleife noch einmal verändert. Rückgabewert ist entweder die Segmentadresse des gefundenen MCBs oder $FFFF_{16}$, falls `adr` in keinem Speicherblock lag.

```
WORD get_owner_mcb (void far *adr, struct T_MCB far **mcb_ptr))
{ struct T_MCB far *tmp;               /* temp. MCB-Zeiger        */
  WORD  para;                          /* Segmentadresse ADR      */
  char  last;                          /* letzte MCB-Kennung      */

  para     = PARA (adr);
  *mcb_ptr = getlol () -> mcb_anchor;
  tmp = NULL;
  do
  { last = (*mcb_ptr) -> flag;
    if ((para >  GET_SEG (*mcb_ptr)) &&
        (para <= GET_SEG (*mcb_ptr) + (*mcb_ptr) -> paras))
    { tmp = *mcb_ptr;
    };
    *mcb_ptr = MAKE_FARPTR (GET_SEG (*mcb_ptr)+(*mcb_ptr) -> paras+1,
                            GET_OFF (*mcb_ptr));
  } while ((last != 'Z') && (tmp == NULL));
  mcb_ptr = tmp;
  return ((tmp != NULL) ? tmp -> owner : 0xFFFF);
};
```

Funktion "Get Owner (MCB)":
`WORD get_owner_mcb (void far *adr, struct T_MCB far **mcb_ptr)`

Aufrufparameter:

`adr`	zu suchende FAR-Adresse
`mcb_ptr`	*Adresse* des FAR-Zeigers

Seiteneffekte:

`mcb_ptr`	falls gefunden: MCB, zu dem `adr` gehört; sonst unbestimmt

Rückgabewert:
$FFFF_{16}$: nicht gefunden; sonst: Segmentadresse MCB;

Tabelle 3.12: `get_owner_mcb`: Ermittle für Adresse zugehörigen MCB

3.5 Verwaltung externer Speicher

Als externe Speicher werden alle Speichermedien bezeichnet, auf die der Prozessor keinen unmittelbaren Zugriff über Adreß- und Datenbus hat. Dazu zählen Schreib-/Lesespeicher wie Festplatten und Disketten sowie nur-Lese-Speicher wie z.B. CD-ROM-Laufwerke. MS-DOS unterscheidet nach der Übertragungsart die zwei Gruppen zeichenorientierte und blockorientierte Geräte.

3.5.1 Master- und Partition-Boot-Record

Das Kernel verwaltet die externen Speichermedien Diskette und Festplatte in Form von logischen Speicherblöcken. Alle Zugriffe auf Dateien werden mit Hilfe der Verwaltungsdaten des Dateisystems in Zugriffe auf logische Sektoren transformiert. Die Abbildung von logischen Sektoren auf die tatsächliche Speicherposition übernimmt der für das angesprochene Laufwerk zuständige Gerätetreiber. Dieser benötigt Informationen über die Organisation des Datenträgers, um die es im folgenden Text geht.

Partition Boot Record. Der erste Sektor jedes logischen Laufwerks, der *Partition Boot Record* (PBR), enthält Informationen über Parameter des Datenträgers wie Anzahl der Sektoren, Sektoren pro Spur, Anzahl der Seiten (Oberflächen), Lage des Wurzelverzeichnisses und anderes mehr (Tab. 3.13). Bei einem Diskettenwechsel liest das Betriebssystem zuerst diese Information, den *BIOS Parameter Block* (BPB), ein, und orientiert sich daran bei allen weiteren Zugriffen. Außerdem kann der Bootsektor auch Code enthalten, der beim Urladen automatisch gelesen und ausgeführt wird (s. 3.5.3 “Der Urladevorgang”).

Logische Laufwerke. Festplatten können softwaremäßig in mehrere *logische Laufwerke* aufgeteilt sein. Diese *Partitions* sind genau wie Disketten organisiert. Im Fall von Diskettenlaufwerken sind physikalisches und logisches Laufwerk quasi identisch, denn Disketten können nicht partitioniert werden. Jedes logische Laufwerk hat beginnend mit Sektor 0 folgenden Aufbau:

- Partition Boot Record
- Tabelle der Sektorzuordnung zu Dateien (FAT; **F**ile **A**llocation **T**able)
- evtl. Kopien des FAT
- Wurzelverzeichnis
- Datenbereich (mit Unterverzeichnissen)

Welche Datei welche Sektoren belegt, ist im FAT festgehalten. Damit die Anzahl der Einträge in der Zuordnungstabelle nicht zu groß wird, faßt MS-DOS mehrere Sektoren, zumeist zwei oder vier, zu einem *Cluster* (engl.: Gruppe) zusammen. Dies ist die kleinste Zuordnungseinheit, die vergeben werden kann. Im Verzeichniseintrag einer Datei steht die Nummer des ersten Clusters. Zeiger im FAT verweisen auf den jeweils nächsten Cluster oder signalisieren das Ende der Datei.

Offset	Typ	Bedeutung
00_{16}	code	Sprung zum Bootprogramm
03_{16}	text	OEM-Name und Versionsnummer
$0B_{16}$	word	Bytes pro Sektor
$0D_{16}$	word	Sektoren pro Cluster
$0E_{16}$	word	reservierte Sektoren (ab Sektor 0)
10_{16}	byte	Anzahl der FATs
11_{16}	word	Anzahl der Einträge im Wurzelverzeichnis
13_{16}	word	Gesamtanzahl Sektoren
15_{16}	byte	Media Descriptor Byte
16_{16}	word	Sektoren pro FAT
18_{16}	word	Sektoren pro Spur
$1A_{16}$	word	Anzahl der Leseköpfe
$1C_{16}$	dword	Anzahl der versteckten Sektoren
20_{16}	dword	Gesamtanzahl der Sektoren (ab hier: DOS 4.0+)
24_{16}	byte	physikalische Laufwerksnummer
25_{16}	byte	reserviert
26_{16}	byte	erweiterte Boot-Signatur (29_{16})
27_{16}	dword	Datenträger-Kennung (Nummer)
$2B_{16}$	text	Datenträger-Kennung (Text)
36_{16}		reserviert
$3E_{16}$	code	Bootprogramm

Tabelle 3.13: Aufbau PBR (Bootsektor)

Master Boot Record. Festplatten verfügen im Unterschied zu Disketten über eine übergeordnete Organisationsebene, die ein physikalisches Laufwerk in mehrere logische Laufwerke, die Partitions, unterteilt. Die Aufteilungsinformation ist im ersten physikalischen Sektor eines Festplattenlaufwerks, dem *Master Boot Record* (MBR; Tab. 3.14), enthalten und wird nur einmal beim Bootvorgang ausgewertet. Der MBR kann nicht über Kernel-Aufrufe gelesen oder geschrieben werden, weil er nicht Bestandteil einer Partition ist.

Jeder der Partitioneinträge ist wie in Tabelle Tab. 3.15 dargestellt aufgebaut. Durch die Angabe des Start- und Endsektors wird die Ausdehnung einer Partition vollständig definiert. Der Typ gibt zusätzliche Informationen zur Datenorganisation an. Ein besonderer Fall ist der der erweiterten Partition, die exakt wie ein Festplattenlaufwerk mit eigenem MBR aufgebaut ist.

Jede Partition stellt sich Programmen, die Dateifunktionen verwenden, als völlig eigenständiges Laufwerk dar, das mit dem logischen Sektor null beginnt. Auf BIOS-Ebene ist von dieser Aufteilung nichts mehr zu bemerken, denn die logische Struktur der Partitionierung wird auf Gerätetreiberebene in die korrekten Angaben für physikalisches Laufwerk, Zylinder, Kopf und Sektor transformiert. Dies bedeutet natürlich

Offset	Typ	Bedeutung
000_{16}	code	Master Boot Program
$1BE_{16}$	struct	1. Partitionseintrag
$1CE_{16}$	struct	2. Partitionseintrag
$1DE_{16}$	struct	3. Partitionseintrag
$1EE_{16}$	struct	4. Partitionseintrag
$1FE_{16}$	word	Boot-Signatur ($AA55_{16}$)

Tabelle 3.14: Aufbau MBR (Master Boot Record)

Offset	Typ	Bedeutung
00_{16}	byte	Bootflag (00_{16}: nicht bootbar; 80_{16} bootbar)
01_{16}	byte	Kopf (Start)
02_{16}	word	Zylinder/Sektor (Start)
04_{16}	byte	Typ (0: frei; 1: 12-Bit FAT; 4: 16-Bit FAT; 5: extended; 6: huge (DOS 4.0+))
05_{16}	byte	Kopf (Ende)
06_{16}	word	Zylinder/Sektor (Ende)
08_{16}	dword	erster Sektor der Partition
$0C_{16}$	dword	Größe der Partition in Sektoren

Tabelle 3.15: Aufbau Partitions-Eintrag

auch, daß Kontrollfunktionen auf BIOS-Level nicht ohne weiteres bestimmen können, auf welche Partition eines Laufwerks gerade zugegriffen wird. Dies ist nur indirekt über die Verwaltungsinformationen im MBR möglich, mit deren Hilfe sich feststellen läßt, in welcher Partition der durch die Aufrufparameter spezifizierte Sektor liegt.

Funktionen zur Sektorbearbeitung. Funktionen zum Lesen und Schreiben von logischen Sektoren bietet das Kernel mit den Interrupts 25_{16} "Absolute Disk Read" (Tab. A.19) und 26_{16} "Absolute Disk Write" an. Die tatsächliche Organisation in Zylinder, Kopf und Sektor spielt keine Rolle, die Sektoren sind mit 0 beginnend fortlaufend durchnumeriert. So entspricht z.B. der logische Sektor 0 auf Diskette dem physikalischen Sektor an Position Zylinder 0, Kopf 0, Sektor 1. Bei der Benutzung beider Funktionen ist zu beachten, daß nach dem Aufruf ein Wort, das von der Funktion gerettete Flagregister, auf dem Stack verbleibt. Dieses ist zu entfernen, um ein unkontrolliertes Wachstum des Stack zu verhindern.

Physikalische Sektoren werden mit dem BIOS-Disk-Interrupt 13_{16} bearbeitet (Tab. A.24). Alle Diskettenoperationen, auch die Aufrufe der Interrupts 25_{16} und 26_{16}, werden über diesen Interrupt abgewickelt. Durch dessen Kontrolle verfügt man über eine grobe, aber sehr effektive Möglichkeit, Dateien und Daten auf einem bestimmten Laufwerk zu schützen. Aber Vorsicht: Manche Geräte wie RAM-Disks und Magnetbandlaufwerke werden von speziellen Gerätetreibern verwaltet, die nicht auf BIOS-Funktionen zurückgreifen müssen und können. Zugriffe auf diese Laufwerke sind für den Lauscher am

Disk-Interrupt unsichtbar und können nur auf Kernelebene abgefangen werden.

Ein Nachteil ist, daß der Interrupt 13_{16} wirklich nur physikalische Laufwerke unterscheidet. Falls ein physikalisches Laufwerk in mehrere logische Laufwerke unterteilt ist, können Aufrufe für verschiedene Partitions eines Laufwerks nicht ohne weiteres erkannt werden. Beispiel: Das Festplattenlaufwerk "0" sei in die logischen Laufwerke "`C:`", "`D:`" und "`E:`" unterteilt. Trotzdem ist die Laufwerksnummer für jede beliebige Operation, die eine dieser Partitions betrifft, immer 80_{16}. Die Umsetzung findet offensichtlich durch Gerätetreiber auf höherer Ebene statt.

Turbo-C stellt die Funktionen

```
int biosdisk (int cmd, int drive, int head, int track,
              int sector, int nsects, void *buffer)          und
int absread (int drive, int nsects, int lsect, void *buffer)
```

zur Verfügung, die dem Zugriff auf BIOS- und Kernelebene entsprechen (Tab. A.24, A.19). Aus den Parametern ist gut ersichtlich, daß auf Kernelebene die reale Organisation des Datenträgers keine Rolle mehr spielt. Zur Beachtung: Bei einigen der durch `cmd` ausgewählten Funktionen haben manche Parameter z.T. eine andere oder keine Bedeutung. Für die Funktionen Lesen, Schreiben und Verifizieren ist der angegebene Prototyp korrekt.

```
#define BDM_RESET   0x00              /* Reset                  */
#define BDM_STATUS  0x01              /* Status abfragen        */
#define BDM_READ    0x02              /* lesen                  */
#define BDM_WRITE   0x03              /* schreiben              */
#define BDM_VERIFY  0x04              /* verifizieren           */
#define BDM_FORMAT  0x05              /* formatieren            */
#define BDM_PARAMS  0x08              /* Parameter abfragen     */
```

Beispiel "ReadPart"

Fangen wir mit einigen Hilfskonstruktion an. Die Makros `GET_TRACK` und `GET_SECTOR` extrahieren Zylinder und Sektor aus der BIOS-üblichen Kombinationsangabe, die ein Wort umfaßt. Die Sektornummer besteht aus 6, die Zylindernummer aus 10 Bits. Um es kompliziert zu machen, hat Microsoft die höchsten 2 Bits der Zylindernummer in den ungenutzten höchsten zwei Bits des Sektor-Bytes untergebracht.

```
#define GET_TRACK(a)  (((a & 0x00C0) << 2) | (a >> 8))
#define GET_SECTOR(a) (a & 0x003F)
```

Das Hauptprogramm versucht, beginnend mit dem ersten Festplattenlaufwerk, die Partitionsdaten des Systems auszugeben. Die Suche wird abgebrochen, falls der MBR nicht eingelesen werden kann, weil entweder das Laufwerk nicht existiert oder ein Fehler aufgetreten ist.

```
void main ()
{ BYTE pdrive = 0x80;
  BYTE ldrive = 0;

  /* solange der MBR gelesen werden kann                        */
  while (read_part_data (&ldrive, pdrive, 0, 0, 1) == 0)
  {  pdrive++;                        /* naechstes phys. Laufwerk*/
  };
  return;
};
```

Die eigentliche Arbeit leistet die Funktion `read_part_data`, die aber auch nicht sehr aufwendig geraten ist. Konnte der MBR erfolgreich in die Struktur `T_BOOTSEC` eingeladen werden, gibt die Routine Informationen zu allen vier Einträgen aus. Eine Besonderheit stellt die erweiterte Partition dar, die exakt wie ein physikalisches Laufwerk aufgebaut ist. Wird eine erweiterte Partition angetroffen, ruft sich die Funktion elegant und platzsparend rekursiv mit den Positionsdaten des MBRs der erweiterten Partition auf.

```
/* Aufbau Partitionseintrag fuer T_BOOTSEC                          */
struct T_PART
{ BYTE  bootflag;                       /* 0x80: bootbar ("aktiv") */
  BYTE  start_head;                     /* Startkopf               */
  WORD  start_combi;                    /* Startzylinder, -sektor  */
  BYTE  type;                           /* Partitions-Typ          */
  BYTE  end_head;                       /* Endkopf                 */
  WORD  end_combi;                      /* Endzylinder, -sektor    */
  DWORD sec_before;                     /* Sektoren vor Partition  */
  DWORD sec_in;                         /* Sektoren in Partition   */
};

/* Aufbau PBR/MBR (kombiniertes Format)                             */
struct T_BOOTSEC
{ BYTE  jump[3];                        /* Code: "JMP <Urladeprg.>"*/
  char  oem[8];                         /* Herstellername          */
  WORD  bps;                            /* Bytes pro Sektor (512)  */
  BYTE  spc;                            /* Sektoren pro Cluster    */
  WORD  res_sectors;                    /* Anzahl reservierte Sekt.*/
  BYTE  fats;                           /* Anzahl FATs             */
  WORD  root_entries;                   /* Anzahl Eintraege Wurzel.*/
  WORD  sectors;                        /* Anzahl Sektoren         */
  BYTE  mdb;                            /* Media Descriptor Byte   */
  WORD  spf;                            /* Sektoren pro FAT        */
  WORD  spt;                            /* 3.0: Sektoren pro Spur  */
  WORD  heads;                          /* 3.0: Anzahl der Koepfe  */
  DWORD hid_sectors;                    /* 3.0: Anz.verst. Sektoren*/
  DWORD tot_sectors;                    /* 4.0: ges. Anz. Sektoren */
  BYTE  phys_drive;                     /* 4.0: phys. Laufwerksnr. */
  BYTE  res1[1];                        /* 4.0: reserviert         */
  BYTE  ext_signature;                  /* 4.0: erw. Boot-Signatur */
  DWORD volume_id;                      /* 4.0: Kennzahl           */
  char  volume_label[11];               /* 4.0: Datentraegername   */
  BYTE  res2[8];                        /* 4.0: reserviert         */
  BYTE  res3[0x0180];                   /* Urladeprogramm          */
  struct T_PART part[4];                /* Partitions-Information  */
  WORD  signature;                      /* "0xAA55" falls bootbar  */
};

int read_part_data (BYTE *ldrive, BYTE pdrive,
                    WORD track, BYTE head, BYTE sector)
{ struct T_BOOTSEC mbr;                 /* Puffer fuer MBR         */
  WORD  i;                              /* Partition-Zaehler       */
  BYTE  type;                           /* Partition-Typ           */
  BYTE  error;                          /* Resultat Leseoperation  */

  if ((error = xbiosdisk (BDM_READ, pdrive, head, track, sector, 1,
       (BYTE far *)&mbr)) != 0)
  { return (error);
  };
```

```
>> hier zusaetzliche Funktionen zur MBR-Bearbeitung einfuegen      <<

  printf ("master boot record at cylinder %d, head %d, sector %d\n",
          track, sector, head);
  for (i = 0; i < 4; i++)
  { type = mbr.part[i].type;
    printf ("entry nr. %d\n", i);
    switch (type)
    { case 0x00:
        printf ("empty\n"); break;
      case 0x01:
        printf ("primary DOS-Partition, 12-bit-FAT\n"); break;
      case 0x02:
        printf ("XENIX-Partition\n"); break;
      case 0x04:
        printf ("primary DOS-Partition, 16-bit-FAT\n"); break;
      case 0x05:
        printf ("extended DOS-Partition -> recursing\n");
        read_part_data (ldrive, pdrive,
          GET_TRACK  (mbr.part[i].start_combi),
          mbr.part[i].start_head,
          GET_SECTOR (mbr.part[i].start_combi));
        printf ("returned\n");
        break;
      case 0x06:
        printf ("'huge' DOS 4.0-Partition\n"); break;
      default :
        fprintf (stderr, "unknown type: %d\n", mbr.part[i].type);
    };

    /* normale Partition?                                          */
    if ((type == 0x01) || (type == 0x04) || (type == 0x06))
    {
>> zeige Daten ueber Partitionseintrag an                          <<
>> hier zusaetzliche Funktionen zur PBR-Bearbeitung einfuegen      <<
      (*ldrive)++;                     /* inkr. log. Laufwerksnr. */
    };
  };
  return (0);
};
```

3.5.2 Funktionen zur Dateibearbeitung

Das DOS-Kernel realisiert die Dateidienste und bildet Dateioperationen in Lese- und Schreibvorgänge mit logischen Sektoren ab. Ein guter Teil der Kernel-Funktionen befaßt sich mit Dateioperationen, die nicht durch den ANSI-Standard für "C" abgedeckt sind, von denen wir aber trotzdem einige benötigen. Dies sind in erster Linie Funktionen, die sich mit Dateiattributen und der Struktur des Dateisystems befassen.

FILE- und Handle-Funktionen. Turbo-C stellt zwei Funktionsgruppen zur Dateibearbeitung zur Verfügung. Die ANSI-"C"-Funktionen beginnen mit dem Buchstaben '`f`' und benutzen die Struktur `FILE`, um eine geöffnete Datei zu referenzieren. Funktionsprototypen finden sich in der Datei `stdio.h`. Die MS-DOS-Funktionen hingegen verwenden eine Kennzahl, das *Handle* (engl.: Griff, Klinke), um eine Datei zu identifizieren. Die zugehörige `include`-Datei heißt `io.h`.

Handle- und FCB-Funktionen. Aus grauer CP/M-Vergangenheit unterstützt MS-DOS auch noch sog. FCB- (**F**ile **C**ontrol **B**lock) Funktionen, die aber kein hierarchisches Dateisystem unterstützen und lt. Microsoft nicht mehr verwendet werden sollten. Daran halten wir uns bis auf eine Ausnahme, die einen Aufruf betrifft, der uns den Umgang mit Dateinamen erleichtert. Die Funktion

```
char *parsfnm (const char *cmdline, struct T_FCB *fcb, int option)
```

(Parse Filename) zerlegt den in `cmdline` übergebenen Dateinamen in das FCB-Format (Tab. A.4). `option` gibt die Art und Weise der Bearbeitung vor und sollte auf 00_{16}[5] gesetzt sein. Ein evtl. im Dateinamen enthaltener Pfad wird ignoriert. Name und Erweiterung (ohne trennenden Dezimalpunkt) liegen separat und mit Leerzeichen aufgefüllt, aber ohne abschließendes 0-Byte vor. Das Jokerzeichen '`*`' wird in eine Folge von '`?`' umgesetzt, was z.B. bei Funktionen zum Vergleich von Dateinamen hilfreich ist. Von der FCB-Struktur sind für uns nur die Felder `name` und `ext` von Bedeutung.

```
struct T_FCB
{ BYTE  drive;                          /* log. Laufwerk ("A:" = 1)*/
  char  name[8];                        /* Name                    */
  char  ext[3];                         /* Erweiterung             */
  WORD  block;                          /* Blocknummer             */
  WORD  rec_size;                       /* Datensatzgroesse        */
  DWORD file_size;                      /* Groesse (in Bytes)      */
  WORD  date;                           /* Datum (kodiert)         */
  WORD  time;                           /* Zeit (kodiert)          */
  BYTE  reserved[8];                    /* reserviert              */
  BYTE  record;                         /* Datensatznummer         */
};
```

Die zu entwickelnden Programme werden sich aus verschiedenen Gründen teilweise der Handle-Gruppe der "C"-Funktionen bedienen. Zum einen erwarten einige Betriebssystemfunktionen z.B. zum Lesen der Dateizeitmarke die Übergabe eines Handles. Zum anderen können speicherresidente Programme keine Dateifunktionen der ANSI-Gruppe verwenden, weil dies immer zu Fehlern führt, die interne Ursachen haben.

Zugriff auf Dateiattribute. *Dateiattribute* sind Daten, die nicht Inhalt der Datei sind, diese aber beschreiben. Dazu zählen Eigenschaften wie Name, Länge, Erstellungsdatum und -zeit sowie die Attribute im Sinne von MS-DOS (`readonly` etc.). Mit den Turbo-C-Funktionen

```
int xgetftime (int handle, struct T_FTIME *ftime)  und
int xsetftime (int handle, struct T_FTIME *ftime)
```

werden Dateidatum und -zeit gelesen bzw. gesetzt (Tab. A.17). Da die Funktionen die Übergabe eines Dateihandles erwarten, muß die Datei zuvor geöffnet werden. Die Struktur `T_FTIME` verwendet Bitfelder, die zeigen, wie die Datums- und Zeitinformation kodiert ist. Hinweis: Die Sekundenangabe erfolgt in Einheiten zu zwei Sekunden. So steht z.B. der Wert 16 für 32 Sekunden. Die Wertebereiche der Bitfelder lassen auch unzulässige Eintragungen zu. Auf diesen Umstand kommen wir noch einmal bei der Entwicklung des Programms "`ChkState`" zu sprechen.

[5] "Alle existierenden Einträge im FCB überschreiben"

```
struct T_FTIME
{ unsigned second:5;                    /* Wort 1, Bits  0 -  4    */
  unsigned minute:6;                    /* Wort 1, Bits  5 - 10    */
  unsigned hour:5;                      /* Wort 1, Bits 11 - 15    */
  unsigned day:5;                       /* Wort 2, Bits  0 -  4    */
  unsigned month:4;                     /* Wort 2, Bits  5 -  8    */
  unsigned year:7;                      /* Wort 2, Bits  9 - 15    */
};
```

Dateisuche. Zur Suche nach Dateien dienen die Funktionen

```
int findfirst (const char *filename, struct T_DTA *dta, int attrib) und
int findnext (struct T_DTA *dta)
```

Zum Einleiten der Suche ist `findfirst` aufzurufen, danach lassen sich weitere Dateinamen mit `findnext` ermitteln (Tab. A.14). `filename` darf die Jokerzeichen '`*`' und '`?`' enthalten, die für eine Folge bzw. ein beliebiges Zeichen stehen. `attrib` legt fest, welche Arten von Dateien in die Suche einbezogen werden sollen. Die Typdefinition `T_DTA` definiert die Struktur des DTAs (**D**isk **T**ransfer **A**rea), über das Daten über die Datei und die aktuelle Suchposition ausgetauscht werden. Datum und Zeit sind wie in `T_FTIME` kodiert. Durch den Autor ermittelte undokumentierte Felder und Sachverhalte stehen in eckigen Klammern "`[...]`".

```
struct T_DTA                            /* Disk Transfer Area      */
{ BYTE  drive;                          /* [log. Laufwerk "A:" = 1]*/
  BYTE  res1[11];                       /* reserviert [alles '?']  */
  BYTE  res2[1];                        /* reserviert              */
  WORD  entry_nr;                       /* [Eintragsnr. im Verz.]  */
  WORD  cluster;                        /* [Verzeichnis-cluster]   */
  BYTE  res3[4];                        /* reserviert              */
  BYTE  attr;                           /* Attribut-Byte           */
  WORD  time;                           /* Zeit (kodiert)          */
  WORD  date;                           /* Datum (kodiert)         */
  DWORD size;                           /* Groesse (in Bytes)      */
  char  name[13];                       /* Name(beendet durch '\0')*/
};
```

Aktueller Pfad. Bei Dateioperationen ohne Angabe von Laufwerk und/oder Verzeichnis wird das aktuelle Laufwerk bzw. das aktuelle Verzeichnis des Laufwerks herangezogen. Das aktuelle Laufwerk wird über die Funktion

```
int getdisk (void)
```

bestimmt (Tab. A.1). Die Numerierung der Laufwerke beginnt dabei mit `A:` = 0. Das aktuelle Verzeichnis wird mit

```
int chdir (const char *path)                bzw.
int getcurdir (int drive, char *directory)
```

bestimmt (Tab. A.9; inklusive Laufwerk) bzw. gelesen (Tab. A.11). Die Laufwerksnummer 0 steht ausnahmsweise für das aktuelle Laufwerk; die weitere Numerierung beginnt mit `A:` = 1.

Besonderheiten: Gerätedateien. Manche Dateinamen wie `con` und `lpt1` sind für zeichenorientierte Geräte reserviert, die über normale Dateifunktionen angesprochen werden können. Der Aufruf

```
int isatty (int handle)
```

(engl. "is a TTY' = "ist ein Fernschreiber [serielles Gerät]") ermittelt, ob es sich bei der durch das Handle referenzierten Datei um ein serielles Gerät handelt. Diese Eigenschaft ist insofern bedeutsam, als daß gewisse Funktionen wie mehrfaches sowie wahlfreies Lesen und Schreiben der Datei nicht möglich sind. Ein Kopierprogramm beispielsweise muß außerdem der Tatsache Rechnung tragen, daß von seriellen Geräten nicht im Binärmodus gelesen werden darf, weil sonst nie das Dateiende = Ende der Eingabe erkannt wird (s.a. 4.4.1 "`AVCopy`").

3.5.3 Der Urladevorgang

MS-DOS besteht aus mehreren Komponenten, die zum Teil in nichtflüchtigem Speicher, dem ROM-BIOS, untergebracht sind und zum Teil erst in den Arbeitsspeicher geladen werden müssen. Letzterer verliert beim Abschalten der Stromversorgung seinen Inhalt und kann durch unglückliche Aktionen anderer Programme überschrieben werden. Aus diesen Gründen muß ein Mechanismus existieren, der beim Einschalten oder Reset des Rechners das Betriebssystem von Diskette oder Festplatte lädt. Der dafür zuständigen Teil des ROM-BIOS trägt die Bezeichnung *Bootstrap Loader*. Für den Namen ist die Münchhausen-Methode des Betriebssystems, sich quasi selbst zu laden, verantwortlich. Das englische Pendant zu "sich an den eigenen Haaren aus dem Sumpf ziehen" lautet "lifting yourself by your own bootstraps" (sich an den eigenen Schnürsenkeln hochheben).

Bei einem *Hardware-Reset* (`reset`-Eingang der CPU wird aktiviert oder der Strom eingeschaltet) führt die CPU automatisch eine interne Initialisierung durch und setzt den Programmzähler auf die Adresse `FFFF:0000`, die Reset-Einsprungstelle. Im Bereich von `F000:0000` bis `F000:FFFF` (höchste Systemadresse) befindet sich zugleich das ROM-BIOS, das 64 kB nichtflüchtigen Speicher umfaßt. An der Reset-Einsprungstelle sind bis zum Adreßraumende noch 16 Bytes Platz für die Reset-Routine und das Datum der ROM-BIOS-Version vorhanden. Da das etwas knapp bemessen ist, steht hier ein Sprung auf die eigentliche Resetroutine.

Diese führt zunächst einen Systemtest durch, um die Konfiguration des Rechners zu ermitteln und die Funktion der einzelnen Komponenten zu überprüfen. Dazu gehört auch der Test bestimmter Speicherbereiche auf die Anwesenheit von Adapterkarten wie z.B. Video- und Festplattencontrollern. Das BIOS durchsucht den Speicherbereich von $\texttt{C8000}_{16}$ bis $\texttt{E0000}_{16}$ in 2 kB-Schritten auf die Kennung $\texttt{AA55}_{16}$ am Anfang jedes Blocks. Im Adapter-ROM folgt dem Kennwort eine Längenangabe und die Initialisierungsroutine des ROMs. Diese wird vom Boot-ROM des Rechners mit einem `call far`-Befehl aktiviert und kehrt mit einem `retf` zur Bootsequenz zurück. Wichtig für Schutzbelange ist die Tatsache, daß zusätzliche ROM-Bausteine *vor* dem unten beschriebenen Bootvorgang aktiviert werden. Viele Schutzsysteme auf Hardwarebasis nutzen die beschriebene Technik.

Nach dem POST (**P**ower **O**n **S**elf **T**est = Selbsttest nach Einschalten) und evtl. Initialisierung der Adapter versucht der Bootstrap-Loader, den Bootblock zunächst von Laufwerk `A:` zu laden. Befindet sich keine Diskette im Laufwerk oder ist kein Laufwerk mit der Bezeichnung `A:` angeschlossen, erfolgt ein zweiter Versuch von der als "aktiv" gekennzeichneten DOS-Partition der Festplatte. Diese unveränderliche, da im ROM-BIOS fixierte Reihenfolge stellt Schutzsoftware vor große Probleme. Eine im Laufwerk `A:` befindliche, evtl. mit einem Boot-Virus infizierte Diskette wird nämlich immer gerne vom Rechner angenommen. Schutzprogramme haben es dagegen schwer, denn sie werden nur durch Laden des Betriebssystems von der Festplatte aktiviert. Doch betrachten wir zunächst den Ladevorgang etwas genauer.

Der Bootstrap-Loader lädt den Bootblock, den ersten physikalischen Sektor der Diskette oder Festplatte, an die Adresse `07C0:0000` und gibt die Kontrolle an diesen

ab. Bis zu diesem Punkt erfolgt der Bootvorgang quasi Betriebssystem-neutral. Das im Bootblock enthaltene Urladeprogramm ist dagegen betriebssystemspezifisch gestaltet und z.B. für MS-DOS, UNIX und OS/2 unterschiedlich. Diese Methode hat den Vorteil, das PCs unter beliebigen Betriebssystemen betrieben werden können.

Unter MS-DOS unterscheiden sich Start von Diskette und Start von Festplatte insofern, als daß der Urlader im MBR erst noch den Urlader der aktiven Partition lädt und ausführt. Dann prüft das Bootprogramm, ob die ersten beiden Dateien im Wurzelverzeichnis mit den versteckten Systemdateien `io.sys` (`ibmbio.com`) und `msdos.sys` (`ibmdos.com`) identisch sind (in Klammern die Dateinamen unter IBMs PC-DOS). Falls dem nicht so ist, wird der Benutzer zum Einlegen einer anderen Diskette und zum Drücken einer Taste aufgefordert. Sind die Dateien vorhanden, lädt das Bootprogramm entweder gleich beide Dateien oder nur die Datei `io.sys` und startet deren Ausführung. Unter Betriebssystemversionen mancher Hersteller liest `io.sys` die Datei `msdos.sys` erst zu einem späteren Zeitpunkt ein.

`io.sys` selbst besteht aus zwei verschiedenen Teilen mit unterschiedlichen Aufgaben. Der erste ist das eigentliche herstellerspezifische BIOS, das eine Reihe von Gerätetreibern für Tastatur und Bildschirm (`con`), Drucker (`lpt*`), serielle Ports (`com*`) und Disketten-/Festplattenlaufwerke enthält. Dazu kommen noch hardwareabhängige Initialisierungsroutinen für diverse Peripheriebausteine. Das RAM-BIOS stellt genormte Schnittstellen zu der vom Hersteller ausgewählten Hardware her. Der zweite Teil von `io.sys`, das Modul `sysinit`, stammt von Microsoft. Es

- stellt die Größe des verfügbaren Arbeitsspeichers fest,
- transferiert sich an das obere Speicherende,
- verschiebt das in der Datei `msdos.sys` enthaltene Kernel an seine endgültige Position oberhalb des BIOS und
- startet dessen Initialisierungsroutine.

Diese baut interne Arbeitsbereiche und Tabellen auf, setzt die Kernel-Interrupts 20_{16} bis $2F_{16}$ und initialisiert die Gerätetreiber. Ab diesem Zeitpunkt steht MS-DOS vollständig zur Verfügung.

config.sys. Erst jetzt kann der Anwender in den Bootvorgang softwaremäßig eingreifen. `sysinit` lädt und interpretiert die im Wurzelverzeichnis befindliche Konfigurationsdatei `config.sys`. Dabei werden optional die Größe bestimmter interner Puffer, die Landessprache und der Name des Kommandointerpreters festgelegt. Am wichtigsten sind die beliebig spezifizierbaren Gerätetreiber, die in das Betriebssystem eingebunden und initialisiert werden. Zuletzt wird der Kommandointerpreter gestartet, der kraft Voreinstellung `command.com` heißt und mit der `shell`-Anweisung in `config.sys` explizit benannt werden kann.

autoexec.bat. Als erstes arbeitet `command.com` die Stapeldatei mit dem Namen `autoexec.bat`, falls im Wurzelverzeichnis vorhanden, ab. Ansonsten wird der Benutzer nach Datum und Uhrzeit gefragt. Jetzt steht der PC dem Anwender für seine Zwecke zur Verfügung.

Aus der Beschreibung des Urladeprozesses ist ersichtlich, daß es mehrere Stellen gibt, an denen in den Ablauf eingegriffen werden kann:

- Bootstrap-Loader (ROM-BIOS, unveränderlich)
- Bootblock (erster phys. Sektor bzw. erster log. Sektor der aktiven Partition)
- Systemdateien (im Wurzelverzeichnis)
- `config.sys` (im Wurzelverzeichnis)
- Kommandointerpreter (evtl. spezifiziert durch `shell` in `config.sys`)
- `autoexec.bat` (im Wurzelverzeichnis)

3.6 Ein- und Ausgabe

Video. Die Ausgabe auf den Bildschirm stellt das wichtigste Mittel dar, um mit dem Anwender zu kommunizieren. Die IBM-PCs sind über die Jahre hinweg mit verschiedenen Grafikkarten ausgestattet worden. Trotz vieler Gemeinsamkeiten wäre die Programmierung recht kompliziert, wenn ein Programm mit jeder Karte und jedem Modus zurechtkommen müßte. Das BIOS trennt den Anwender von der Hardware, indem es über den Interrupt 10_{16} ("Video-Interrupt") Funktionen zur Videoausgabe zur Verfügung stellt. Unabhängig von der Videokarte können bestimmte Modi gesetzt, Zeichen und Grafik ausgegeben sowie der Cursor gesetzt und abgefragt werden. Insgesamt bedient der Video-Interrupt über 50 verschiedene Funktionen und Subfunktionen. Viele Programme programmieren den Videocontroller und schreiben direkt ins Video-RAM, um z.B. die Textausgabe stark zu beschleunigen. Da unsere Programme nur kurze Meldungen ausgeben müssen, verwenden wir die BIOS-Funktionen. Das bringt uns außerdem den wichtigen Vorteil der Hardwareunabhängigkeit ein.

Die Turbo-C-Funktion

```
void gotoxy (int x, int y)
```

setzt den Cursor auf die angegebene Stelle des Bildschirms (Tab. A.20). Basis ist mit den Koordinaten (1; 1) die linke obere Ecke. Die Cursorposition wird mit den Funktionen

```
int wherex (void), int wherey (void)
```

abgefragt. Die Ersatzfunktion

```
void xwherexy (int *x, int *y)
```

in `msdos_s.lib` erledigt die Abfrage in einem Aufruf (Tab. A.21). Das Zeichen an der Cursorposition wird mit den Funktionen

```
writechar (BYTE attr, BYTE chr)       und
int readchar (BYTE *attr, BYTE *chr)
```

geschrieben bzw. gelesen (Tab. A.22, A.23). Die letzten beiden Funktionen sind unter Turbo-C nicht verfügbar, sondern nur in `avsys.lib` enthalten.

Keyboard. Für die Eingabe von Tastatur stehen die Funktionen des "Keyboard-Interrupts" 16_{16} bereit. In Turbo-C wurden die einzelnen Funktionen nicht wie beim Videointerrupt separat implementiert, sondern es existiert eine Funktion

```
int bioskey (int cmd)
```

zum Aufruf aller Dienste. Durch die sparsame Ausstattung mit Übergabeparametern können nicht alle der neun Funktionen verwendet werden, doch reichen die übrigen völlig aus (Tab. A.25):

```
#define BKM_GETKEY  0x00              /* lese Zeichen          */
#define BKM_PEEPKEY 0x01              /* pruefe auf Zeichen    */
#define BKM_FLAGS   0x02              /* lese Sondertasten     */
```

`cmd` bestimmt den Abfragemodus. Modus 0 liest ein anstehendes Zeichen oder wartet auf ein neues Zeichen von der Tastatur. Im Modus 1 wird nur angefragt, ob ein Zeichen ansteht (nein: Rückgabewert 0) und wenn ja, welches. Der Aufruf im Modus 2 schließlich liefert ein Bitmuster, das den Zustand der Sondertasten repräsentiert. Ein gesetztes Bit steht für eine gedrückte Taste; die Reihenfolge von Bit 0 bis Bit 7 ist: Shift rechts, Shift links, Control, Alternate, Scroll Lock, Numkey Lock, Capitals Lock, Insert.

Unsere Programme benutzen nur die Modi 0 und 1, um auf einen Tastendruck des Benutzers zu warten. Sinn der Aktion ist es, die Verwendung der Funktion `getch` zu vermeiden, die eine ganze Kette speicherfressender Module nach sich zieht. Vor allen Dingen bei residenten Programmen spielt der Platzverbrauch durchaus eine Rolle.

3.7 Speicherresidente Programme

Mit Blick auf die angestrebten Kontrollmaßnahmen wenden wir uns nun den speicherresidenten Programmen zu. Diese Programmklasse hat die Eigenschaft, auch nach ihrer Beendigung ständig im Arbeitsspeicher zu verbleiben und auf bestimmte Ereignisse zu reagieren. Dies sind entweder Interruptaufrufe von Anwenderprogrammen oder Dienstanforderungen an das Betriebssystems. Residente Programme sind in der Lage, im Hintergrund unsichtbar für den Anwender Systemfunktionen zu überwachen und gegebenenfalls auf bestimmte Zustände zu reagieren. Damit stellen sie die Methode zur Systemüberwachung schlechthin dar.

MS-DOS kennt zwei Sorten speicherresidenter Programme, die der Benutzer installieren kann: Die Gerätetreiber, die Bestandteil des Betriebssystems sind, und die TSR-Programme, die eine Unterabteilung der Anwenderprogramme darstellen. Beide Typen sind von Einbindung und Aufbau her sehr unterschiedlich und haben verschiedene Vor- und Nachteile, die es gegeneinander abzuwägen gilt.

3.7.1 TSR-Programme

Das Kernel bietet zwei Methoden an, Programme resident im Speicher zu installieren. Die erste, ältere Möglichkeit, führt über den Interrupt 27_{16} und sollte nicht mehr benutzt werden. Der DOS-Funktionsinterrupt 21_{16} bietet die moderne und daher von uns

verwendete Funktion 31_{16} "Terminate and Stay Resident" (TSR) an (Tab. A.7). Als Parameter werden der Rückgabewert des Programms und die Größe des zu reservierenden Speichers in Paragraphs erwartet. Der Anfang des Speicherbereichs ist mit dem Anfang des aktuellen PSPs, d.h. mit dem Anfang des Programms identisch.

Somit ist der Ablauf bei der Installierung wie folgt:

- Programm wird aufgerufen
- Auf bereits bestehende Installation prüfen, ggf. Ausführung abbrechen
- Initialisierung durchführen (Interrupts umsetzen, Datenbereiche reservieren und vorbelegen etc.)
- Speicherbedarf feststellen (vorher evtl. Environment freigeben und Puffer reservieren)
- Programm mit Aufruf Interrupt 21_{16}, Funktion 31_{16} beenden

3.7.2 Gerätetreiber

Wie im vorangegangenen Abschnitt gesehen, ist es relativ einfach, ein Programm im Speicher zu installieren. Wie dieser Abschnitt zeigen wird, ist der Aufwand bei Gerätetreibern ungleich größer. Der Grund, warum wir uns dann trotzdem mit der zweiten Klasse residenter Programme beschäftigen, liegt in der etwas größeren Sicherheit, die die Installation als Gerätetreiber mit sich bringt. Umfangreiche Informationen zur Programmierung von Gerätetreibern finden sich in [13]. Im nächsten Abschnitt werden die Vor- und Nachteile der beiden Konzepte ausführlich diskutiert.

Gerätetreiber werden nicht wie gewöhnliche TSR-Programme gestartet und installiert, sondern über die Konfigurationsdatei `config.sys` beim Bootvorgang in das Betriebssystem eingebunden. Die Datei `config.sys` ist ein gewöhnlicher ASCII-Text mit einer Anweisung pro Zeile. Der Eintrag für einen Gerätetreiber hat die Form

```
device = <vollst. Dateispezifikation> [<parameter>]

Beispiel:
device = c:\dos\emm386.sys 1024
```

Dateistruktur. Gerätetreiberdateien (`*.sys`) haben einen besonderen Vorspann, der Informationen über den Namen, Typ, verfügbare Funktionen und die Adressen der Serviceroutinen enthält. Beim Urladen fügt die Laderoutine von `sysinit` den neuen Gerätetreiber in die sequentiell vorwärts verkettete Liste der bereits geladenen ein.

Gerätetreiberfunktionen werden vom Kernel nach einem relativ komplizierten Verfahren aufgerufen, das gleich zwei Bearbeitungsroutinen erfordert [13]. Für unsere Zwecke sind die normalen Treiberfunktionen irrelevant, da wir "nur" eine residente Plattform erstellen wollen. Trotzdem müssen wir zumindest die Funktion `init` implementieren, die von `sysinit` nach der Installierung aufgerufen wird. Dadurch kommen wir auch um die Kommunikationsfunktionen nicht herum.

Aus dem Gesagten folgt, daß die Implementierung nicht ausschließlich in "C" erfolgen kann. Der besondere Aufbau des Vorspanns und die Art der Treiberaufrufe

erfordern zumindest die teilweise Realisierung in Assembler, am günstigsten in Form eines Gerätetreiber-"C"-Interfaces. Da Zeichentreiber einfacher als Blocktreiber aufgebaut sind, wären diese der Typ unserer Wahl.

Tips. Bücher zum Thema Gerätetreiber erwähnen oft und mit Recht, daß Serviceroutinen des Treibers keine Kernel-Funktionen aufrufen dürfen. Bedeutet dies, daß unser Programm z.B. keine Dateioperationen durchführen darf? Zum Glück nicht, den das Verbot erstreckt sich nur auf die Treiberroutinen, die *vom Kernel* aufgerufen werden und die aus Reentrancy-Gründen nicht wieder Kernelfunktionen in Anspruch nehmen dürfen. Unsere, in bestimmte Interrupts eingeklinkten Kontrollfunktionen sind aber keine Routinen, die im Rahmen eines Treiberaufrufs angesprochen werden. Für sie gilt das Gleiche, was schon im Abschnitt über die TSR-Programme gesagt wurde: Die Kontrollfunktion wird aktiviert, *bevor* der Aufruf das Betriebssystem erreicht.

3.7.3 Vor- und Nachteile

Was sind nun die Vor- und Nachteile der beiden Konzepte? Die Realisierung eines speicherresidenten Programms als Gerätetreiber ist wie gesehen nur mit einem guten Stück Mehraufwand möglich, der sich irgendwie bezahlt machen sollte. Bei unserer Methode, einen zurechtgestutzten Gerätetreiber als residente Plattform zu gebrauchen, besteht der wesentliche Unterschied nur in der Art, in der die beiden Programmtypen geladen werden. Diesen Unterschied gilt es zu untersuchen.

Start von TSR-Programmen. TSR-Programme müssen wie alle Programme gestartet werden, um sie resident zu installieren. Aus Sicherheitsgründen sollte dies nach einem Systemneustart

1. automatisch (ohne Beteiligung des Anwenders),
2. möglichst sicher (ohne Eingriffsmöglichkeit durch den Benutzer) und
3. möglichst früh (vor der Aktivierung anderer Programme)

erfolgen. Der naheliegende Weg für Programme ist ein Eintrag in die Stapeldatei `autoexec.bat`, die von `command.com` automatisch am Ende des Bootprozesses ausgeführt wird. Die Abarbeitung von Stapeldateien kann durch den Benutzer über das Drücken der Tastenkombination Control C oder Control Break abgebrochen werden. Das ist dann ungünstig, wenn mit Aktionen der Anwender gegen das Schutzprogramm zu rechnen ist.

Ein Ausweg bestünde in einer Veränderung von `command.com`, die bewirkt, daß die Shell den Befehl zum Abbruch nicht mehr erkennt oder zumindest die Sicherheitsabfrage beim Benutzer automatisch immer mit "Nein" beantwortet. Weil dies aber keine besonders schöne Lösung ist, betrachten wir doch einmal den Ladevorgang bei einem als Gerätetreiber realisiertem Schutzprogramm, wie es z.B. `F-Driver.sys` des Softwarepakets `F-Prot` darstellt.

Start von Gerätetreibern. Gerätetreiber werden zu einem Zeitpunkt installiert, zu dem das System noch nicht auf Eingaben des Benutzers reagiert oder überhaupt

reagieren kann. Dies ist der wesentliche Vorteil dieser Installationsmethode. Die Aktivierung vor dem Start des Kommandointerpreters ermöglicht außerdem eine Überprüfung desselben auf Veränderungen. Gerade die Shell ist potentieller Infektionsträger und -verbreiter Nummer Eins, da alle Operationen und Dateizugriffe auf Kommandoebene über dieses Programm laufen. Ist `command.com` erst einmal mit einem Virus verseucht, führt praktisch jede Eingabe des Benutzers zur Verbreitung der Infektion.

Ein gemeinsamer Nachteil: Beide Startverfahren sind von Konfigurationsdateien, `autoexec.bat` bzw. `config.sys`, abhängig. Wenn der Benutzer diese verändern kann, ist beim nächsten Systemstart der Schutz dahin. Schutzkonzepte müssen also auch Sorge dafür tragen, daß diese Dateien unveränderlich sind. Dazu bietet sich die Deklaration als `hidden` und `readonly` an, an der sich MS-DOS beim Urladen nicht stört. Werkzeuge zum Verändern von Dateiattributen sind von Festplatte zu verbannen. Ein anderer Weg wäre der aktive Schutz durch ein Watcher-Programm, der keine Manipulationen an Inhalt und Attributen zuläßt.

Fazit: Jede Installationsmethode hat ihre Vorteile, die bei TSR-Programmen in der Einfachheit und bei Gerätetreibern im rel. sicheren Start liegen. Doch auch Gerätetreiber sind nicht unfehlbar; spätestens dann nicht, wenn das Betriebssystem von Diskette geladen wird. Wenn dies geschieht, kann der Rechner sofort mit einem Bootvirus verseucht werden. Durch Urladen von Diskette läßt sich *jedes* Schutzsystem auf Softwarebasis umgehen. Weil der Sicherheitsvorteil der Gerätetreiber verglichen mit dem Mehr an Programmieraufwand nur gering ist, werden wir TSR-Programme als Plattform für unsere Watcher verwenden.

Kapitel 4

Entwicklung der Systemprogramme

> Was wäre, wenn Programmiersprachen Autos wären?
>
> Assembler: Ein Go-Cart ohne Sicherheitsgurte und Überrollbügel. Gewinnt jedes Rennen, wenn es nicht vorher im Graben landet.
>
> C: Ein offener Geländewagen. Kommt durch jeden Matsch und Schlamm, der Fahrer sieht hinterher auch entsprechend aus.
>
> Aus *Happy Computer*

In den vorangegangenen drei Kapiteln ging es um Grundlagen der Softwareanomalien, die Theorie ihrer Abwehr und die Systemprogrammierung unter MS-DOS. Wir haben uns dabei das nötige Wissen über Computerviren und Systeminterna erworben, um erfolgreich Abwehrprogramme entwerfen und erstellen zu können. Dieses Kapitel beschreibt die Umsetzung unserer Erkenntnisse in konkrete "C"- und Assembler-Programme. Dabei sollen alle Programmtypen behandelt oder zumindest angesprochen werden, die im Kapitel "Theorie der Abwehr" beschrieben sind. Resultat ist ein Antivirus-System ("AV-System"), das durchaus nicht nur Lehrcharakter hat, sondern auch in der Praxis sinn- und wirkungsvoll eingesetzt werden kann.

Aufbau des Kapitels. Der erste Entwicklungsschritt, die *Problemstellung*, ist uns bereits aus den Kapiteln "Theorie der Softwareanomalien" und "Theorie der Abwehr" bekannt. Den Anfang macht deshalb die *Anforderungsanalyse*, in der zunächst Forderungen an das Programmpaket formuliert und der Leistungsumfang festgelegt werden, ohne sich Gedanken um die Realisierung zu machen. Anhand der Informationen über Arbeitsweise und Fähigkeiten von Computerviren wird eine Übersicht der sicherheitsrelevanten Kommandos und Funktionen unter MS-DOS erstellt. Mit Hilfe dieser Angaben lassen sich die noch vagen Anforderungen präzisieren. Resultat der

Anforderungsanalyse ist eine funktionsmäßige Beschreibung der notwendigen Abwehrmaßnahmen.

Der *Systementwurf* konkretisiert die Ergebnisse der Anforderungsanalyse und zerlegt die zu leistenden Aufgaben in Systemkomponenten, d.h. zunächst in einzelne Abwehrprogramme. Diese lassen sich in drei Gruppen aufteilen:

1. Im Abschnitt *Prüfprogramme* werden Programme behandelt, die sich weder resident installieren noch in die Arbeit von MS-DOS eingreifen. Dazu gehören Checker und Scanner.
2. Die *Kontrollprogramme* stellen eine Erweiterung bzw. Veränderung des Betriebssystems auf Programmebene dar. Realisiert werden neue externe Kommandos mit Kontrollfunktionen, die interne Befehle ersetzen.
3. Die *speicherresidenten* Programme kontrollieren Systemaufrufe auf beiden Interruptebenen. Alle Watcher basieren auf einer universell einsetzbaren TSR-Plattform und einem Interrupt/"C"-Interface, die wichtige Grundfunktionen realisieren.

Die Beschreibungen der Programme sind nach einem festen Schema gegliedert:

- Problemstellung: Möglichst genaue Schilderung des Problems, ohne Expertenwissen vorauszusetzen.
- Aufgabenbeschreibung: Was genau soll das Programm leisten? Die Problemstellung wird analysiert, Lösungen werden erarbeitet.
- Systemarchitektur: Welche Funktionen werden durch welches Modul realisiert, wie arbeiten die Module zusammen (Schnittstellen, Fehlerbehandlung etc.)?
- Funktionsbeschreibung: Beschreibung der verwendeten Algorithmen, evtl. auch konkret anhand des Quelltextes.
- Hinweise: Zusätzliche Erläuterungen zur Funktion, besondere Kniffe bei der Programmierung, Tips für die Anwendung.
- Diskussion: Wo liegen Schwachstellen, wie sind sie — falls möglich — zu beheben? Wo liegen die Grenzen des Verfahrens, worin bestehen sinnvolle Erweiterungen? Dieser Teil wurde zusammenfassend ins 5. Kapitel ausgelagert.

Doch nun genug der Vorrede — fangen wir an!

4.1 Anforderungsanalyse

Forderungen an das Programmpaket. Eine sorgfältige Anforderungsanalyse ist der Schlüssel für effektive Programmierung und die Erreichung der gesetzten Ziele. Wie sehen diese überhaupt aus? Das Softwarepaket soll

- verhindern, daß verseuchte Programme in das zu schützende Rechnersystem eindringen, d.h. durch menschliches Zutun oder automatisch ablaufende Vorgänge ins System gebracht werden (kontrollierte Isolation)
- passiv vorliegende Viren (= infizierte Programme) detektieren und identifizieren (Scanner)
- Veränderungen an der Struktur des Dateisystems und am Dateibestand aufdecken (rein quantitativ; Checker)
- die durch Verbreitung oder Schadensfunktionen hervorgerufenen inhaltlichen Veränderungen von Dateien durch den Einsatz von (kryptographischen) Prüfsummen und den Vergleich mit Tabellen der korrekten Werte erkennen (Checker)
- dito, aber Veränderung der Dateiattribute (Datum, Zeit, Attribut-Byte und weitere Charakteristika; Checker)
- nicht validierte oder erkennbar verseuchte/veränderte Programme am Start, d.h. Aktivieren des Virus hindern (kontrollierte Isolation)
- aktiven Viren die residente Installation und Übernahme von Interrupts verwehren (Watcher)
- aktive Computerviren am Vorhandensein im Speicher und an Aktivitäten erkennen (Scanner, Watcher)
- die unbefugte Manipulation von Programmen und anderen Dateien aktiv verhindern (Watcher)
- die durch aktive Viren ausgelösten Schadensfunktionen abwenden (z.B. Formatieren der Festplatte; Watcher).

Das ist ein ganzer Forderungskatalog. Die Übersicht ist nach dem Grad der Aktivität der Viren in mehrere Verteidigungslinien gestaffelt. Während der letzte Punkt sich sozusagen des Viren-GAUs[1] annimmt, befaßt sich die erste Forderung mit einem sauberen System und dem Versuch, diesen Zustand zu erhalten. Es sollte festgehalten werden, daß eine Blockierung des Virus an einer der vorgeschobenen Stellungen wünschenswert ist. Mit zunehmender Aktivität des Virus wird es nämlich für das Antivirusprogramm immer schwieriger, den Schutz des Systems aufrecht zu erhalten. Moderne Stealth-Viren beispielsweise tricksen nicht nur Dateiscanner aus, sondern können sogar Suchprogrammen, die auch den Speicher überprüfen, Schwierigkeiten bereiten. Fazit: Ist das Virus erst einmal aktiv, ist es für Schutzmaßnahmen bereits zu spät.

Anforderungen des Virus. Der Forderungskatalog spiegelt auch die typischen Aktivitäten eines Virus wider. Die drei Hauptaufgaben der Computerviren sind:

1. Verbreitung
2. Manipulation des Systems (zur Tarnung, effektiveren Arbeit etc.)
3. Schädigung

[1] **G**rößter **a**nzunehmender **U**nfall

Von der Verbreitung abgesehen gilt die Aufstellung auch für Trojaner. Für die angeführten Aktivitäten benötigt jede Softwareanomalie bestimmte Funktionen und Ressourcen des Betriebssystems, die wir uns genauer ansehen wollen.

Zu 1: Verbreitung. Bei der Verbreitung sind zwei Phasen zu unterscheiden. Beim Transport auf den Rechner sind noch relativ komplexe Programme (zum Kopieren, Empfang via DFÜ etc.) und der Anwender im Spiel. Ab der Aktivierung handelt das Virus dagegen völlig selbständig und verwendet elementare Funktionen des Betriebssystems. Dies gilt auch für die nächsten zwei Punkte. Zur passiven und aktiven Verbreitung tragen folgende Aktionen und Funktionen bei:

- Transport auf den Rechner (über wechselbare Datenträger und Kopierprogramme, DFÜ, Tastatur; mit/ohne Unterstützung des Anwenders),
- Ausführung (Start durch Anwender oder andere Programme, bes. im Urladeprozeß),
- Optional: Residente Installation (als TSR, durch Reservierung von Speicher oder Verminderung des TOM[2]),
- Aktive Suche nach Programmen oder alternativ Einklinken in Interrupts (Infect on Open),
- Datei manipulierbar machen (Attribute verändern),
- Datei öffnen, lesen, manipulieren, schreiben (→ Infizieren).

Am Transport verseuchter Programme ist zwangsläufig der Anwender beteiligt, denn zu diesem Zeitpunkt ist das Virus noch ein totes Stück Code. Einen wichtigen Unterschied macht es, ob der Anwender das Virus absichtlich auf den Rechner bringt oder ob dies aus Unachtsamkeit oder Unwissenheit geschieht. Im ersten Fall muß das Schutzprogramm so ausgelegt sein, daß es auch gezielten Attacken widersteht. Einfaches Beispiel ist ein als Datendatei getarntes Programm, das auf die Festplatte kopiert und dort umbenannt wird.

Zu 2: Manipulation des Systems (Tarnung). Schritt 2 ist nicht unbedingt erforderlich bzw. unter jedem Betriebssystem möglich. MS-DOS-Rechner machen es Softwareanomalien sehr leicht, in Betriebssystemabläufe einzugreifen. Ist das einmal geschehen, kann sich der Anwender auf keine Reaktion des Systems mehr verlassen; der Eindringling hat den Rechner völlig unter Kontrolle. Unter komplexeren Betriebssystemen ist eine solche Manipulation von Funktionen nicht oder nur über die Ausnutzung von Fehlern und Hintertüren in der Schutzsoftware möglich. Zur Tarnung gehören:

- Abfangen von Interrupts (Vermeidung von Fehlermeldungen, Umleitung auf saubere Kopie bei Stealth-Viren, “Desinfect on Open”),
- Anlegen versteckter Dateien (Spawning Viruses, komplexe Trojaner),
- Anlegen versteckter Datenblöcke (bes. für die Speicherung des sauberen Urladeblocks bei Stealth-BSIs),

[2] **T**op **O**f **M**emory

- Anlegen versteckter Verzeichnisse (Trojaner wie AIDS),

- Veränderung der Systemzeit (um Logdaten unbrauchbar zu machen).

Zu 3: Schädigung. Nicht jedes Virus versucht, seine Umgebung zu schädigen, während dies bei Trojanern meist die Hauptfunktion ist. Es lassen sich vier Schadensarten unterscheiden:

- Verbrauch von Ressourcen (Rechenzeit, Speicherplatz; schon durch Anwesenheit gegeben),

- Löschung von Daten,

- Verfälschung von Daten,
- Beschädigung der Hardware.

Der Begriff "Daten" ist dabei weit gefaßt. In Frage kommen einzelne Dateien, die Dateisystemstruktur, Datenverwaltungsinformation, ganze Datenträger, der Inhalt des Arbeitsspeichers (RAM) und die Konfigurationsdaten (CMOS-RAM).

Während MS-DOS bis zu diesem Punkt eher neutral betrachtet wurde, geht es in den folgenden Abschnitten konkret um MS-DOS in seiner Eigenschaft als Betriebsmittel für Softwareanomalien. Durch unsere Untersuchungen wissen wir bereits, welche Funktionen ein Virus für seine Aktivitäten benötigt. Es soll nun untersucht werden, welche Funktionen und Eigenschaften von MS-DOS konkret an Virenattacken beteiligt sind und welche Rolle die Systemstruktur dabei spielt. Für jede Ebene von MS-DOS wird geprüft, welche Dienste durch Softwareanomalien in Anspruch genommen werden. Erkannte Sicherheitslücken sind durch Maßnahmen organisatorischer Natur und durch Systemprogramme mit Überwachungsfunktionen zu schließen.

4.1.1 Kommandoebene

Dieser Abschnitt umfaßt Kontrollen auf Programm- oder Kommandozeilenebene, der obersten Schicht der MS-DOS-Hierarchie. Um die im Abschnitt 2.7.1 "Schutzzonen und Kontrollpunkte" beschriebenen Transport- und Umbenennungs-Operationen durchzuführen, sind bestimmte DOS-Kommandos erforderlich, die in der zweiteiligen Tabelle C.1 im Anhang C.1 aufgelistet sind.

Sicherheitsrelevante Befehle. Betrachten wir die Tabelle der MS-DOS-Kommandos unter dem Gesichtspunkt "Transport von Dateien". Gesucht ist eine Liste der sicherheitsrelevanten Befehle, die durch neue Kommandos mit Kontrollfunktionen zu ersetzen sind. Tabelle 4.1 faßt das Ergebnis des Selektionsvorgangs nach Funktionsgruppen geordnet zusammen.

Die Transportfunktionen treten zum Teil versteckt auf. `format` überträgt, falls mit der Option `/s` aufgerufen, den Bootblock und die Systemdateien `io.sys`, `msdos.sys` und `command.com` auf die formatierte Diskette. Diese wird dadurch urladefähig, d.h. MS-DOS kann von dieser Diskette gestartet werden. Ein relativ unbekannter Befehl ist `sys`, der die Systemdateien nachträglich auf Diskette oder Festplatte überträgt. Voraussetzung für die Anwendung ist, daß bei der Formatierung entsprechend Platz reserviert wurde (z.B. mit `format /b`).

Kommandos der beiden anderen Gruppen sind von Bedeutung, falls der Anwender gegen die Schutzprogramme arbeitet. Mit `attrib` lassen sich schreibgeschützte Programme wieder beschreibbar machen. `ren` bzw. `rename` kann zum Tarnen eines Programms vor dem Kopieren dienen. Die Kommandos `join` und `subst` bewirken, daß der Herkunftsort einer Datei u.U. verschleiert wird (s.a. C.2 "Assign, Join, Subst"). Mit `date` und `time` lassen sich Systemdatum und -zeit verändern und dadurch Logdaten unbrauchbar machen.

Befehl	Typ	Funktion
Informationstransport		
`backup`	E	Dateien sichern
`copy`	I	Dateien kopieren (und umbenennen)
`diskcopy`	E	Diskette kopieren
`format`	E	Diskette formatieren
`replace`	E	Erweiterte `copy`-Version
`restore`	E	Mit `backup` gesicherte Dateien wieder einlesen
`sys`	E	Betriebssystemdateien kopieren
`xcopy`	E	Verbesserte `copy`-Version
Veränderung Dateiattribute		
`attrib`	E	Dateiattribute anzeigen/ändern
`ren(ame)`	I	Dateien umbenennen
Begünstigung		
`append`	E	Pfad für Daten-Dateien definieren
`date`	I	Datum anzeigen/ändern
`join`	E	Laufwerk als Unterverzeichnis anhängen
`subst`	E	Verzeichnis als Laufwerk ansprechen
`time`	I	Zeit anzeigen/ändern

Tabelle 4.1: Sicherheitsrelevante Kommandos unter MS-DOS

Wie bereits angesprochen, unterscheidet die MS-DOS-Shell `command.com` zwischen internen Befehlen und normalen Programmen, zu denen auch die externen Kommandos zählen. Dieser Sachverhalt ist insofern bedeutsam, als daß sicherheitskritische externe Kommandos einfach durch Löschen der Programmdatei entfernt werden können. Falls die Befehle für den bestimmungsgemäßen Betrieb des Rechners notwendig sind, müssen sie durch funktional ähnliche Programme mit Kontrollfunktionen ersetzt werden.

Wenn auch interne Befehle Belange der Systemsicherheit betreffen, steht man vor dem Problem, daß die Überwachungssoftware die Eingabe des Benutzers prüfen müßte, noch bevor diese zum Kommandointerpreter gelangt. Zwei Ansätze zur Lösung sind denkbar. Entweder sorgt man dafür, daß der zu überwachende Befehl extern wird, oder man rüstet den Kommandointerpreter mit Kontrollfunktionen aus. In beiden Fällen sind neue Programme zu erstellen, welche die Aufgaben der sicherheitsrelevanten MS-DOS-Befehle übernehmen.

4.1.2 Interruptebene

Funktionsaufrufe über Software-Interrupts, wie sie bei MS-DOS- und BIOS-Aufrufen verwendet werden, sind einfach und flexibel zu handhaben. Dem aufrufenden Programm muß die absolute Einsprungadresse der Routine nicht bekannt sein, was besonders bei Änderungen im Betriebssystem (z.B. neues Release) von Vorteil ist. Außerdem können bestehende Aufrufe leicht abgefangen, erweitert oder verändert werden, indem einfach der betreffende Aufrufvektor auf die neue Serviceroutine umgesetzt wird. Speziell für

Kontrollprogramme (Watcher), aber natürlich auch für Softwareanomalien (z.B. Infect on Open- und Stealth-Viren) ergeben sich hier Möglichkeiten, in Betriebssystemabläufe einzugreifen.

Sicherheitsrelevante Interrupts. Die Tabellen C.3 und C.4 im Anhang zeigen, welche Interrupts überhaupt zur Verfügung stehen bzw. durch welche Teile von MS-DOS sie bedient werden. Auch bei den Interrupts müssen wir die relevanten aussieben. Wichtig sind vor allen Dingen der BIOS-Disk-Interrupt 13_{16} und der DOS-Funktions-Interrupt 21_{16}. Alle Operationen, die Dateien, deren Attribute und die Struktur des Dateisystems betreffen, laufen über den Interrupt 21_{16}, der zur Kernelebene gehört. Das Kernel (über die Gerätetreiber) und Anwenderprogramme wiederum nutzen die sektororientierten Dienste des Interrupts 13_{16}.

Durch die Aufteilung der Betriebssystemfunktionen in Kernel- und BIOS-Aufrufe können und müssen Kontrollprogramme auf zwei unterschiedlich abstrakten Ebenen aktiv werden. Mit dem Disk-Interrupt befinden sich alle externen Datenträger unter unserer Kontrolle, die nicht von speziellen Gerätetreibern verwaltet werden. Der Schutz auf dieser Ebene ist eher allgemein (lesen, schreiben) und umfaßt gleich ein ganzes Laufwerk. Die Kernelfunktionen erlauben uns eine feiner granulierte und präzisere Überwachung auf Dateiebene. Objekt (die Datei) und Operation sind genau bestimmt und können für spezifischere Kontrollen ausgewertet werden. Außerdem ist die Überwachung von Operationen auf Laufwerken möglich, die durch spezielle Gerätetreiber verwaltet werden (RAM-Disk etc.).

Sicherheitsrelevante Funktionen (Kernel). Das DOS-Kernel bietet mit dem Interrupt 21_{16} eine Vielzahl von Funktionen, vor allen Dingen zur Dateibearbeitung, an. Dazu kommen Aufrufe für die Ein-/Ausgabe auf best. Geräten, die interne Speicherverwaltung und Funktionen für den Zugriff auf Systemdaten wie Datum und Zeit, Interruptvektoren und andere mehr. Es erscheint sinnvoll, die Fülle der Systemaufrufe hinsichtlich der Arbeitsweise von Computerviren in übersichtliche, funktionale Gruppen zu unterteilen.

Das Ergebnis unserer Auslese ist in Tabelle 4.2 wiedergegeben. Am wichtigsten ist der Aufruf zum Starten von Programmen, weil dadurch potentiell ein Virus aktiviert wird. Hier greifen Schutzmaßnahmen wie Überprüfung der Integrität und kontrollierte Isolation an. Die zweite Gruppe enthält Funktionen, die sich ganz konkret mit der Manipulation von Dateien beschäftigen. Besonders interessant sind Zugriffe auf Programmdateien, die auf dieser Ebene im Gegensatz zum BIOS-Level einfach zu erkennen sind. Normalerweise schreiben nur Compiler und Kopierprogramme ausführbare Dateien. Eine Warnmeldung zur Beurteilung durch den Anwender oder eine automatische Überprüfung würde Programme gegen illegale Manipulationen schützen.

Die anderen Funktionsgruppen befassen sich mit sekundären Effekten wie Tarnung und Schädigung. Manipulationen an Dateiname und -attributen könnten ihre Ursache in versuchter Täuschung des Schutzprogramms (Tarnung) oder in Vorbereitungen zur Dateiveränderung (Infektion, Schädigung) haben. Infect on Open- sowie Stealth-Viren müssen sich resident installieren und Interrupts übernehmen, um so in die

[3] ASCII-String, durch Null- (Zero-) Byte beendet.

Nr.	ab Ver.	Beschreibung
Ausführung von Programmen		
$4B_{16}$	2.0	execute program (ASCIIZ[3])
Ein-/Ausgabe von Dateiinhalten		
$0F_{16}$	1.0	open file (FCB)
$3D_{16}$	2.0	open file (ASCIIZ, handle)
$6C_{16}$	4.0	extended open file (ASCIIZ, handle)
15_{16}	1.0	sequential write (open FCB)
22_{16}	1.0	random write (open FCB)
28_{16}	1.0	random block write (open FCB)
40_{16}	2.0	write file or device (handle)
Lesen/Schreiben von Dateiattributen		
17_{16}	1.0	rename file (special FCB)
56_{16}	2.0	rename file (ASCIIZ)
43_{16}	2.0	get or set file attributes (ASCIIZ)
57_{16}	2.0	get or set file date and time (handle)
Manipulation der Dateisystemstruktur		
13_{16}	1.0	delete file (FCB)
41_{16}	2.0	delete file (ASCIIZ)
16_{16}	1.0	create file (FCB)
$3C_{16}$	2.0	create file (ASCIIZ, handle)
$5A_{16}$	3.0	create temporary file (ASCIIZ, handle)
$5B_{16}$	3.0	create new file (ASCIIZ, handle)
$6C_{16}$	4.0	extended open file (ASCIIZ, handle)
39_{16}	2.0	create directory
$3A_{16}$	2.0	delete directory (ASCIIZ)
Auffinden von Dateien		
11_{16}	1.0	find first file (FCB)
12_{16}	1.0	find next file (FCB)
$4E_{16}$	2.0	find first file (ASCIIZ)
$4F_{16}$	2.0	find next file (ASCIIZ)
Beeinflussung des Systems		
25_{16}	1.0	set interrupt vector
$2B_{16}$	1.0	set date
$2D_{16}$	1.0	set time
31_{16}	2.0	terminate and stay resident
48_{16}	2.0	allocate memory block
$4A_{16}$	2.0	resize memory block

Tabelle 4.2: Relevante Funktionen DOS-Funktionsinterrupt 21_{16}

Arbeit des Betriebssystems einzugreifen. Konventionelle Softwareanomalien hingegen suchen aktiv nach Wirtsprogrammen und Dateien zur Manipulation. Durch Kontrolle der Suchfunktionen können bestimmte Dateien ausgefiltert und dadurch unsichtbar gemacht werden. Logdaten lassen sich über Beeinflussung der Systemuhr verfälschen; über die Einträge ist dann kein Rückschluß mehr auf den Zeitpunkt einer Operation möglich. Viren und Trojaner können Schaden durch Löschen von Dateien und Verzeichnissen anrichten. Andererseits legen manche Softwareanomalien versteckte Dateien (Spawning Viruses) und Verzeichnisse (AIDS-Trojaner) an, die Programmcode oder Daten enthalten können.

Sicherheitsrelevante Funktionen (BIOS). Die über den Interrupt 13_{16} aufrufbaren Funktionen betreffen physikalische Laufwerke. Objekte sind einzelne durch Laufwerk, Zylinder, Kopf und Sektornummer bestimmte Sektoren. Tabelle C.5 im Anhang C.1 gibt eine Übersicht über die einzelnen Funktionen. Uns interessieren besonders die Aufrufe, mit denen Daten verändert werden können. Dazu gehören die Operationen zum Schreiben von Sektoren und Formatieren von Zylindern.

Das auf BIOS-Ebene nur physikalische Laufwerke und einzelne Sektoren unterschieden werden, hat für die Sicherheitssoftware Vor- und Nachteile. Zwar kann der Zweck eines Zugriffs nicht oder nicht genau ermittelt werden, denn außer Laufwerk, Position des Sektors und Art der Operation (Lesen, Schreiben, Formatieren) stehen keine Parameter zur Analyse zur Verfügung. Es kann sich dabei ebenso um das Lesen oder Schreiben von Daten zur Dateiverwaltung wie zur Dateiverarbeitung handeln. Dafür sitzt das Kontrollprogramm an der Quelle des Geschehens. Laufwerksbezogener Schutz läßt sich einfach und umfassend realisieren.

Alles in allem sind eine Reihe von Funktionen auf verschiedenen Ebenen von MS-DOS zu überwachen. Es ist deshalb ein Watcher vorzusehen, der sich in die relevanten Interrupts einklinkt, die Aufrufparameter auswertet und je nach Sachlage die Operation zuläßt oder abbricht. Die zu schützenden Objekte sind in unterschiedlich fein aufgelöste Bereiche unterteilbar (Tab. 4.3).

Bereich	Ebene	Funktionen
globale Rechte	Kernel	ändern der Systemzeit
Partition-Rechte	BIOS	schreiben, lesen von Urladeinformation/Sektoren allgemein; formatieren von Spuren
Verzeichnis-Rechte	Kernel	anlegen, löschen, durchsuchen
Dateirechte	Kernel	schreiben, lesen, ausführen, suchen
Datei-"Pflichten"	Kernel	überprüfen der Integrität

Tabelle 4.3: Schutzbereiche eines Watchers

4.1.3 Der Urladevorgang

Ein spezielles Kapitel stellt der Urladevorgang dar, der eine besondere Form des Programmstarts ist (s.a. 3.5.3 "Der Urladevorgang"). Beim Urladen werden eine Reihe von Daten und Programmen automatisch geladen, interpretiert oder direkt ausgeführt. Wann immer dies der Fall ist, besteht auch für Softwareanomalien die Möglichkeit, aktiv zu werden. Urladeviren sind ein besonderes Problem, da sie vor jedem anderen Programm außer der Bootstrap-Routine des ROM-BIOS ausgeführt werden. Insbesondere Stealth-BSIs können ihre Tarnmaßnahmen installieren, bevor Antivirus-Programme die Systemkontrolle übernehmen.

Es muß deshalb verhindert werden, daß das Betriebssystem von einem externen Datenträger urgeladen werden kann. Soviel schon vorab: Ohne Änderung der Urladesoftware (ROM-BIOS) läßt sich kein echter Schutz erzielen.

4.1.4 Qualität der Schutzmaßnahmen

Schutz des Systems. Perfekter Schutz ist nach Cohen nur bei perfekter Isolation möglich. Das bedeutet, daß jegliche Programmerstellung, wegen der Allgemeinheit der Interpretation sogar jegliche Dateneingabe, verboten ist. Computer im militärischen Bereich kommen noch am ehesten an diese Forderung heran. Andere Anwendungen erfordern aber den Austausch von Informationen mit der Außenwelt. Die Forderung an die Undurchdringlichkeit der Abwehrmaßnahmen wird deshalb je nach Anwendungsgebiet unterschiedlich ausfallen. Aus den genannten Gründen werden wir unsere Ansprüche auf ein Erschweren der Verseuchung herunterschrauben müssen. Grund dafür sind nicht zuletzt wirtschaftliche Erwägungen, denn besserer Schutz ist i.d.R. auch mit mehr Aufwand verbunden.

So benötigt der Anwender "zu Hause" nur einen Warnmechanismus, für dessen Funktionsfähigkeit er Sorge tragen wird. In einem Betrieb oder Rechenzentrum dagegen ist mit mutwilliger Verseuchung und Arbeit gegen die Schutzsoftware zu rechnen. Ein Zwischenfall verursacht meist Kosten, die durch Wiederbeschaffung oder Restaurierung der Daten entstehen. Dazwischen liegt z.B. das Rechenzentrum einer Hochschule, in dem das "versehentliche" Einspielen und Starten von Programmen unterbunden werden soll. Eine Überwindung des Schutzsystems hat keine gravierende Folgen, stellt aber u.U. einen Verstoß gegen die Vorschriften für die Nutzung der Rechner dar, der entsprechend geahndet werden kann. Dieser Gedanke folgt §202a des Strafgesetzbuches, der das Ausspähen von Daten nur dann unter Strafe stellt (stellen kann!), wenn der Rechner mit Schutzvorrichtungen ausgerüstet ist.

Selbstschutz. Ein wichtiger Punkt, der besonders berücksichtigt werden muß, ist der Start des Wächterprogramms. Es müssen folgende Anforderungen, die dem Selbstschutz dienen, erfüllt sein:

1. Sicherer Start. Der Startvorgang darf nicht vom Benutzer unterbrochen werden können, wie es z.B. mit `Ctrl` `Break` möglich ist, während die Stapeldatei `autoexec.bat` abgearbeitet wird.

2. Umgehung vermeiden. Der Watcher muß als erstes Programm nach den Systemprogrammen gestartet werden, damit nicht ein anderes Programm Interrupt-Vektoren auf sich umsetzen kann, bevor das Wächterprogramm die Kontrolle erhält.

Die zu ergreifenden Schutzmaßnahmen sind von der Realisierung der Kontrollprogramme (TSR-Programm, Gerätetreiber) und der Ausstattung des Rechners abhängig. Steht ein schreibgeschützter Server zur Verfügung, ergeben sich keine Probleme. Auf einem normalen Rechner im Stand Alone-Betrieb sind Schwierigkeiten durch den Urladevorgang vorprogrammiert. Selbst bei APCs, die mit einem besonderen ROM-BIOS ausgestattet sind, stellen die Konfigurationsdateien `config.sys` (Gerätetreiber) und `autoexec.bat` (TSR-Programme) eine Schwachstelle dar. Diese müssen unveränderbar sein. Die Ausführung von Stapeldateien wie `autoexec.bat` kann zudem vom Anwender abgebrochen werden. `command.com` müßte deshalb so verändert werden, daß dies nicht mehr möglich ist.

4.2 Systementwurf

Die geplanten Schutzmaßnahmen müssen, wie oben gezeigt, Operationen auf Programm- und Interruptebene überwachen und ggf. verhindern. Die Teilung der potentiellen Gefahren macht eine Teilung der Schutzmaßnahmen notwendig. DOS-Kommandos für den Dateitransport werden durch eigene Programme mit Kontrollfunktionen ersetzt. Dazu kommen Programme, die auf Anfrage den Zustand des Systems und der externen Speicher überprüfen. Der residente Teil ist im Hintergrund aktiv und schaltet sich in relevante Kernel- und BIOS-Interrupts ein. Ein Schutz gegen das Urladen von Diskette muß auf Hardware-Ebene realisiert werden. Sowohl die Schutzfunktionen als auch die Selbstschutzfunktion des Wächterprogramms dürfen nicht zu leicht außer Kraft zu setzen sein.

Damit ist eine gewisse Systemarchitektur bereits vorgegeben, die praktisch identisch mit dem Aufbau von MS-DOS ist. Die folgenden Abschnitte befassen sich mit Maßnahmen auf den einzelnen Hierarchieebenen. Es werden bereits einzelne Programme, ihre Aufgaben und ihre prinzipielle Funktion definiert. Den Anfang macht die Erläuterung der Systembibliotheken, auf denen das AV-System aufbaut.

4.2.1 Bibliotheken

Funktionen, die von mehreren Programmen benötigt werden, lagert man sinnvollerweise in eine Softwarebibliothek (engl. library) aus. Vorausgesetzt, daß die Schnittstellen sorgfältig dokumentiert wurden, können diese Funktionen fix und fertig in neue Programme übernommen werden und vereinfachen so die Programmerstellung erheblich. Zum AV-System gehören die beiden Bibliotheken `msdos_s.lib` und `avsys.lib`, die beide für das Speichermodell `SMALL` ausgelegt sind. Diese sind nicht nur Grundlage der Programme des AV-Systems, sondern können auch für eigene Programme und nicht nur

unter "C" eingesetzt werden. Voraussetzung ist lediglich die Verwendung des korrekten Speichermodells und der "C Calling Convention" für Funktionsaufrufe.

msdos_s.lib enthält MS-DOS-spezifische Funktionen, die nicht Bestandteil von ANSI-"C" sind. Manche Compiler wie Turbo-C enthalten zwar bereits entsprechende Bibliotheksfunktion, aber nicht jeder arbeitet mit diesem Übersetzungsprogramm. Deshalb stellt `msdos_s.lib` eine Reihe von Funktionen bereit, die den Turbo-C-Routinen funktional nachempfunden sind; den Funktionsnamen wurde lediglich ein '`x`' vorangestellt. Parameterübergabe und Arbeitsweise sind identisch, soweit es für unsere Zwecke erforderlich ist. Die Quelltexte für `msdos_s.lib` sind übrigens ein gutes Beispiel für den Aufruf von Betriebssystemroutinen von Hochsprachen aus. Funktionsbeschreibungen und -prototypen für Assembler- und "C"-Aufrufe finden sich im Anhang A.3.

avsys.lib ist eine Zusammenstellung recht unterschiedlicher Funktionen, die nur zum Teil speziell für das AV-System entworfen worden sind. Die meisten Routinen lassen sich auch für andere oder ähnliche Zwecke einsetzen, ohne an bestimmte Programme gebunden zu sein. Die Bibliothek setzt sich aus sechs Funktionsgruppen zusammen, die im Anhang A.4 beschrieben sind.

Anmerkungen. Wegen der großen Anzahl von Bibliotheksfunktionen und Programmen ist es unmöglich, alle Quelltexte oder auch nur einen größeren Teil davon abzudrucken. Allein `avsys.lib` enthält 41 Objektdateien, die 49 Funktionen exportieren; bei `msdos_s.lib` sind es 26 Funktionen. Die 15 Programme, die diese Funktionen benutzen, sind z.T. selbst recht umfangreich. Da Systemprogrammierung ganz ohne praktische Beispiele aber etwas zu trocken ist, mußte eine Auswahl getroffen werden. Es wurden deshalb keine Funktionen abgedruckt, die in eine der folgenden Kategorien fallen:

- Triviale Funktionen, die aufgrund der Funktionsbeschreibung einfach nachzuempfinden sind (Beispiel `strcpyu`: String bis zu einem bestimmten Zeichen kopieren).
- Routinen zum Aufruf von Betriebssystemfunktionen, die nur den Inhalt von Variablen in Register transportieren und umgekehrt (z.B. alle in `msdos_s.lib` enthaltenen Funktionen).

Mit Sicherheit dabei sind Funktionen, die

- einen wesentlichen Beitrag zur Realisierung eines Konzeptes leisten und die für die Funktion eines Programms von zentraler Bedeutung sind,
- so komplex sind, daß eine Besprechung des Quelltextes erforderlich ist,
- stellvertretend für andere Funktionen des gleichen Typs eine bestimmte Vorgehensweise verdeutlichen (z.B. das Interrupt-"C"-Interface),
- Algorithmen verwenden, die evtl. nicht jedem zur Verfügung stehen (Beispiel: Berechnung von CRC-Prüfsummen).

Auf der Programmdiskette befinden sich natürlich **alle** (englisch kommentierten) Quelltexte sowie die fertig übersetzten Programme und Bibliotheken.

Aufrufparameter, Seiteneffekte und Rückgabewerte sind für jede Funktion in einer kurzen Übersicht zusammengestellt (Tab. 4.4). Für den Aufruf von Betriebssystemfunktionen existiert jeweils eine kombinierte Beschreibung: ein Teil für den Aufruf aus Assembler heraus und ein Teil für die Realisierung als "C"-Funktion. Aus diesen Angaben kann sich jeder, der das 3. Kapitel durchgearbeitet hat, den Assembler-Quellcode für die "C"-Funktion herleiten.

Funktion "voller Funktionsname":
ggf. "C"-Prototyp

Aufrufparameter:
`Parameter` Bedeutung

Seiteneffekte:
`Parameter` Inhalt nach Funktionsende *oder*

Erläuterungen zur Funktion

Rückgabewert:
Bedeutung

Tabelle 4.4: Standardaufbau Funktionsbeschreibung

Die meisten Funktionsbeschreibungen wurden in den Anhang A ausgelagert, um die Besprechung der Programme übersichtlich zu halten. Falls Sie also im Text auf eine Funktion stoßen, die nicht vor Ort besprochen wird, finden sie die nötigen Informationen im Anhang A.3 "MSDOS_S.LIB" oder A.4 "AVSys.LIB". Die meisten Routinen der MS-DOS-Bibliothek sind daran zu erkennen, daß der Funktionsname mit einem '`x`' beginnt. Die Beschreibung der AV-Systembibliothek enthält ein Register, aus dem ersichtlich ist, welche Funktion wo besprochen wird.

4.2.2 Prüfprogramme

Die Prüfprogramme sind konventioneller Bauart, d.h. es handelt sich um keine speicherresidenten Programme. In die hier besprochene Kategorie fallen Scanner und Checker, die manuell vom Benutzer zu starten sind. Diese Programme sind außerdem MS-DOS-unspezifisch, überwachen also keine Kommandos oder Funktionen des Betriebssystems.

ChkSys (Scanner). "Check System" ist ein Scanner, genauer: eine Scan-Plattform, die verschiedene Funktionen tragen kann, die dateiweise arbeiten. In unserer Version ist die Identifizierung von Viren nicht vorgesehen. Warum nicht, ist das nicht die primäre Aufgabe eines Scanners?

Anfang 1992 gab es bereits über 1000 (Tausend) verschiedene Viren, und täglich werden es mehr. Um brauchbare Signaturen zur Suche zu erhalten, muß jedes Virus verfügbar sein und per Reverse Engineering analysiert werden. Mittlerweile ist das

Ziel, dabei aktuell zu bleiben, nur noch durch gemeinschaftliche Aktivitäten verschiedener Antivirusforscher auf der ganzen Welt zu erreichen. Z.T. arbeiten ganze Gruppen hauptberuflich und durch elektronische Netze miteinander verbunden an der Erforschung neuer Viren. Davon abgesehen, daß dem Autor nur wenige Viren zur Verfügung stehen[4], wäre ein Scan-Programm bereits bei Erscheinen des Buches hoffnungslos veraltet. Aus diesen Gründen werden alternative Funktionen zur Suche nach verdächtigen Dateien verwendet.

Seal und ChkSeal (Checker). "Seal" (engl.: Siegel) ist ein spezieller Checker, der Prüfsummen über einzelne Dateien, ganze Datenträger (Disketten, Wechselplatten) oder die Urladeinformation eines Datenträgers berechnet. Nützlich ist `seal` vor allen Dingen beim Versenden von Daten per Briefpost oder Datennetze. Die Nachricht wird vom Absender elektronisch versiegelt, indem dieser die Prüfsumme berechnet und dem Empfänger übermittelt. Der Empfänger berechnet erneut die Prüfsumme und vergleicht die beiden Werte. Sind sie gleich, wurde die übersandte Information sehr wahrscheinlich nicht verändert. Das Programm `Validate` von McAfee Associates zur Prüfsummenerzeugung über Dateien arbeitet auf die beschriebene Weise.

"Check Seal" dient zur Selbstüberprüfung von Programmen beim Start, so daß diese bei Manipulationen vor sich selbst warnen können. Das Prinzip ist das gleiche wie bei `Seal`: Die aktuelle Prüfsumme wird mit einem Referenzwert verglichen. Nur sind diesmal Prüfalgorithmus und -information im Programm selbst untergebracht. Das Programmsiegel wird nach der Kompilierung von einem Konfigurationsprogramm an die Programmdatei angefügt und bei Bedarf auch wieder entfernt.

ChkState (Checker). Das Programm "Check State" überprüft das Dateisystem anhand einer Referenzliste. Gegenstand der Überwachung sind der Datei*bestand* (gelöschte oder hinzugekommene Dateien und Verzeichnisse) und die Datei*inhalte* und -attribute (Verletzung der Integrität). Die Ergebnisse eines Prüflaufs werden in Form eines Reports ausgegeben, so daß Veränderungen leicht erkennbar sind. Die Referenzliste ist so aufgebaut, daß ein einfacher und schneller Zugriff auf eine bestimmte Datei möglich ist. Das Watcher-Programm `AVWatchI` wird sich dieser Liste zur Überprüfung der Programmintegrität bedienen.

4.2.3 Neue Kommandos

Die Schutzmaßnahmen zur kontrollierten Isolation greifen nicht nur auf Interrupt-, sondern auch auf Programmebene an. Um z.B. den Transport von verseuchten Programmen von Diskette auf Festplatte zu verhindern, müssen diverse MS-DOS-Kommandos mit Kontrollfunktionen ausgerüstet werden. Wie schon angesprochen, wirft diese Forderung bei internen Kommandos Probleme auf.

Prinzipiell wäre es möglich, einen eigenen Kommandointerpreter zu schreiben. Um den Aufwand möglichst gering zu halten, könnte dieser als "Filter" ausgeführt werden, der nur zulässige Operationen an `command.com` weiterleitet. Auf diese Weise bliebe die Programmierarbeit auf die Kontrollfunktionen beschränkt. Noch einfacher

[4] Durch Begegnungen der dritten Art...

wäre es, wenn `command.com` die kritischen internen Kommandos nicht mehr erkennen würde. Eigene Programme könnten deren Aufgaben übernehmen. Um interne Befehle zu "externisieren", muß `command.com` geeignet verändert werden.

Damit die Anzahl der zu ersetzenden Kommandos möglichst gering bleibt, wird aus Gruppen gleichartiger Befehle wie `copy`, `xcopy`, `replace` etc. ein Kommando ausgewählt, das für den Betrieb ausreicht. Die nach unseren Untersuchungen zu entfernenden Befehle sind:

`copy`, `xcopy`, `replace`, `backup`, `restore` → Ersatz durch `AVCopy`
`ren`, `rename` → Ersatz durch `AVRename`
`attrib`, `fdisk`, `format`, `join`, `subst`, `sys` → ersatzlose Streichung

Die besondere Bedeutung von `join` und `subst` für die korrekte Funktion der Watcher-Programme wird im Anhang C.2 untersucht.

AVCopy. Als Ersatz für alle Kommandos mit Transportfunktion dient das Programm `AVCopy`, das dem Kommando `copy` nachgebildet ist, ohne allerdings alle (und selten genutzte) Optionen zu implementieren. Ziel ist ja nicht die Realisierung eines luxuriösen Kopierprogramms, sondern es soll aufgezeigt werden, wie man Kontrollmaßnahmen implementiert. Diese überwachen

- den Transport von Daten- und Programmdateien zwischen Laufwerken und Verzeichnissen.
- die Umbenennung von Programmdateien (Tarnung).
- die Änderung des Dateityps (ausführbar/nicht ausführbar).

AVRename ersetzt das interne Kommando `ren` bzw. `rename`. Durch die Erweiterung der Funktion ist das Umbenennen von Verzeichnis zu Verzeichnis möglich, falls dabei das Laufwerk nicht gewechselt wird. Auf diese Weise können Dateien zwischen Verzeichnissen eines Laufwerks ohne zeitaufwendiges Kopieren und Löschen bewegt werden. `AVRename` kennt die gleichen Rechte wie `AVCopy`, von der Kontrolle des Transports zwischen Laufwerken einmal abgesehen.

4.2.4 Residente Programme

Alle speicherresidenten Programme des Paketes bauen auf einer einheitlichen TSR-Plattform auf, die universell einsetzbar ist. `Std_TSR` realisiert die grundlegenden Funktionen Installation erkennen, Installieren und Deinstallieren. Vorgesehen ist außerdem die Kommunikation mit anderen Programmen. Der Anwender kann an den dafür vorgesehenen Punkten Quelltext für eigene ISRs einklinken und so mehrere Interrupts mit einem Programm überwachen. Zu `Std_TSR` gehört ein in Assembler geschriebenes Interrupt-"C"-Interface, das die Implementation von Kontroll-ISRs in "C" statt Assembler ermöglicht.

AVWatchG (Globale Rechte). Globale Rechte betreffen das ganze System und jedes Programm, das darauf abläuft. `AVWatchG` ist der einfachste der vier Watcher und dient uns als Einstieg in die Verwirklichung von Kontrollfunktionen auf Interruptebene. Die Rechte sind bereits im Quelltext festgelegt und können nur bei der Kompilierung verändert werden. Überwacht werden die Kernel-Funktionen zum Ändern von Systemdatum und -zeit. Bei einem Schreibzugriff wird keine Änderung durchgeführt und der erfolgreiche Abschluß der Operation simuliert.

AVWatchI (Prüfung Integrität). `AVWatchI` überprüft die Integrität von zu startenden Programmen. Dazu klinkt sich der Watcher in die entsprechende Kernel-Funktion ein und ermittelt aus den Parametern den vollständigen Namen des Programms. Die aktuell berechnete Prüfsumme wird mit dem Eintrag in der von `ChkState` angelegten Referenzliste verglichen. Diese Kontrolle kann auf andere Dateieigenschaften wie Datum, Zeit, Attribute und Länge ausgedehnt werden.

Bei einer Abweichung verhält sich `AVWatchI` wie das bereits vorgestellte Betriebssystem S3 von Cohen. Der Anwender kann die Ausführung abbrechen lassen oder das Programm trotzdem starten und ggf. die Signatur auf den neuesten Stand bringen. Bei der Kompilierung ist ein besonderer Modus konfigurierbar, bei dem der Benutzer keine Einflußmöglichkeit hat, sondern der Aufruf immer mit einem simulierten Fehler beendet wird. Dieser Modus ist für Anwendungen gedacht, bei denen mit Gegenmaßnahmen der Benutzer gerechnet werden muß. Jedes der folgenden Programme ist mit dieser Option ausgerüstet.

AVWatchP (Partitionsrechte). Mit `AVWatchP` steigen wir auf die Ebene des BIOS hinab. Gleichzeitig wird es etwas komplexer, denn es sollen nicht nur physikalische Laufwerke unterschieden werden (was recht einfach wäre), sondern einzelne Partitions. Dieses Verfahren ermöglicht die Anlage einer schreibgeschützten und einer beschreibbaren Partition auf *einer* Festplatte, um z.B. Systemprogramme zu schützen und gleichzeitig eigene Programme und Texte erstellen zu können.

Neu ist die Methode zur Übergabe der Partitionsinformationen und der Zugriffsrechte an das residente Programm. Weil diese Daten nur einmal beim Start des Watchers gelesen werden müssen, wird die Leseroutine in `AVConfig` ausgelagert, um den residenten Teil möglichst klein zu halten. Das Konfigurationsprogramm überträgt die Informationen über eine besondere Kommunikationsfunktion an den Watcher.

AVWatchF (Dateirechte). Dieser Watcher ist das mit Abstand komplexeste Exemplar unserer Sammlung. Ziel ist die Überwachung aller Dateizugriffe unter Beachtung einer Liste von Dateirechten. Lohn der Mühe ist ein Programm, das den Funktionsumfang von `Flushot` in etwa umfaßt und in manchen Aspekten sogar übertrifft. Das Rechtekonzept lehnt sich an UNIX an, ohne mit diesem Betriebssystem konkurrieren zu wollen. Unterschieden werden die Rechte zum Lesen, Schreiben und Ausführen von Dateien und zur residenten Installierung von Programmen. Dazu kommen die Pflichten zum Überprüfen der Integrität und zur Protokollierung von zulässigen oder unzulässigen Operationen.

Zur Festlegung der Rechte und Pflichten findet folgendes Verfahren Anwendung:

- Rechteeinträge können auf Dateigruppen, Verzeichnisse und ganze Laufwerke zu-

treffen. Widersprüchliche Rechte führen je nach Voreinstellung zur Zulassung oder Verweigerung des Zugriffs.

- Es muß nicht für jede Datei ein Rechteeintrag vorhanden sein. Je nach Voreinstellung wird der Zugriff in diesem Fall immer als zulässig oder als unzulässig bewertet.

Der Zugriff auf die Rechtedatei muß wegen ihres Umfangs zur Laufzeit erfolgen und kann nicht wie bei `AVWatchP` ausgelagert werden. Eine Funktion von `AVConfig` sorgt für die Umsetzung der Rechtedatei von einem komfortableren Textformat in das kompakte Format für `AVWatchF`.

4.2.5 Hilfsprogramme

LogRep (Log-Report). Die von den Watchern erzeugten Logdaten werden in einer komprimierten Form abgelegt, um den Aufwand für die Schreibroutine im residenten Teil möglichst gering zu halten. Die Umsetzung in eine verständliche Textform erfolgt durch `LogRep`. Mögliche Erweiterungen sind Retrieval-Funktionen, um z.B. nach bestimmten Operationen in einem bestimmten Zeitraum suchen zu können.

AVConfig (AV-System Configuration). Die Aufgaben von `AVConfig` wurden bereits durch die vorangegangen Beschreibungen der anderen Programme definiert. Der Vollständigkeit halber sei hier noch einmal eine Übersicht angegeben:

- Anfügen/Entfernen Programmsiegel für `chk_seal`,
- Einlesen und Übertragen der Partitionsdaten und -rechte zu `AVWatchP`,
- Übersetzung der Rechtedatei `f_rights.raw` in die maschinenfreundliche Form `f_rights.lst`,
- Deinstallierung von Watchern.

Dazu kommen zwei kleinere Programme, die bereits als Übungen im Kapitel über die Systemprogrammierung entwickelt wurden. Diese können als "Toolbox für den Virenjäger" dienen.

- "Read MCB-List" (`ReadMCB`): Ausgabe der MCB-Liste mit Angabe des belegten Adreßbereichs und des Besitzers. Für die Suche nach illegalen speicherresidenten Programmen.
- "Read Partition Information" (`ReadPart`): Ausgabe der MBRs und PBRs aller Festplattenlaufwerke oder einzelner Diskettenlaufwerke.

4.3 Prüfprogramme

4.3.1 Check System (ChkSys)

"Check System" ist das erste unserer konventionellen, d.h. nicht speicherresidenten Programme. Es geht um die Forderung Nummer zwei des Aufgabenkataloges: "passiv vorliegende Viren (= infizierte Programme) detektieren [und identifizieren]". Aus den schon dargelegten Gründen arbeitet `ChkSys` nicht wie ein gewöhnlicher Scanner, sondern macht den Anwender auf Dateien, die durch bestimmte Merkmale auffallen, aufmerksam.

Problemstellung. Manche Softwareanomalien legen versteckte Dateien und Verzeichnisse an, um Daten zwischenzuspeichern oder weil ihr Funktionsprinzip dies erfordert (Trojaner wie "AIDS" bzw. Spawning Viruses). Mit einem gesetzten `hidden`- oder `system`-Flag markierte Dateien tauchen beim Auflisten mit `dir` nicht auf und werden z.B. bei der Ausführung von `del *.*` "übersehen". Mit normalen MS-DOS-Kommandos sind solche Dateien und Verzeichnisse nicht zu entdecken oder zu löschen. Ebenfalls gebräuchlich ist die Verwendung von besonderen Leerzeichen, wie sie durch Drücken der Tastenkombination [Alt] [2] [5] [5] [5] entstehen, zur Verschleierung von Dateinamen.

Einige Computerviren benutzen die Zeitmarke der Datei als "ist infiziert"-Kriterium. So ist es z.B. möglich, in das Feld für die Sekundenangabe die Werte für 60 und 62 Sekunden einzutragen. Da `dir` nur Stunden und Minuten anzeigt, bleibt diese Manipulation ebenfalls vor dem Benutzer verborgen.

Die Systemsicherheit wird aber nicht nur durch Softwareanomalien bedroht. So stellen viele Programme Funktionen zur Verfügung, welche die Fähigkeiten der MS-DOS-Kommandos umfassen und bei weitem übertreffen. Beispiele dafür sind PC-Tools, Norton Utilities und andere Werkzeuge. Falls für den geplanten Einsatz eines Rechners diese Leistungen nicht erforderlich sind, sollten derartige Programme aus dem System entfernt werden ("Need to Have"-Prinzip).

Zu den genannten Gefahrenpunkten kommt eine weitere Schwierigkeit hinzu. Falls die Dateistruktur weit verzweigt ist und im System viele Dateien gespeichert sind, ist die manuelle Suche nach verdächtigen Dateien sowohl zeitaufwendig als auch fehlerträchtig.

Aufgabenbeschreibung. `ChkSys` sucht das gesamte Dateisystem nach verdächtigen Dateien und Verzeichnissen ab und meldet in einem Report alle besonderen Vorkommnisse. Eine Datei oder Verzeichnis ist verdächtig, wenn

- sie die Attribute `readonly`, `hidden` oder `system` trägt (was z.T. auch für die Systemdateien gilt),
- der Name Steuerzeichen oder ungewöhnliche Textzeichen enthält,
- die Zeitmarke in Datum oder Zeit illegale Werte aufweist,
- der Name in einer "schwarzen Liste" enthalten ist,

[5] [Alt] halten, auf Ziffernblock 255 eingeben, [Alt] loslassen

- die Dateilänge gleich 0 ist.

Systemarchitektur. Es ist eine Scan-Plattform `scan_dir` zu entwickeln, mit deren Hilfe Dateibestände durchsucht werden können. Parameter der Suche sind Startverzeichnis, Suchmaske (Name und Attribute), Rekursionstiefe und Funktionen zur Verzeichnis- bzw. Dateiverarbeitung. Durch die Übergabe von Zeigern auf die Bearbeitungsfunktionen wird die Suchroutine universell einsetzbar.

`ChkSys` setzt auf dieser Scan-Plattform auf und fügt ergänzend Kontroll- und Reportfunktionen hinzu, die von `scan_dir` aktiviert werden. Durch diese Bauweise reduziert sich der Programmieraufwand für `ChkSys` auf die eigentlichen Kontrollfunktionen. Das Hauptprogramm wertet lediglich die Aufrufparameter aus, führt einige Initialisierungen durch und ruft die Suchfunktion auf.

Mit `chksys.lst` kann eine "schwarze Liste" von Dateien angegeben werden, deren Anwesenheit auf dem System unerwünscht ist. Jede Zeile enthält genau einen Eintrag, der auch Jokerzeichen, aber keine Pfade enthalten darf. Stimmt der Name einer aufgefundenen Datei mit einem Eintrag der Liste überein, wird dies im Report vermerkt (Abb. 4.1).

Abbildung 4.1: Ein-/Ausgabe `ChkSys`

Funktionsbeschreibung: scan_dir. Das Suchverhalten der Scan-Plattform wird durch vier Parameter gesteuert. `path` bezeichnet den Startpfad, der mit einem Backslash enden muß. Die Suchmaske `mask` darf Jokerzeichen enthalten und wählt die zu suchenden Dateien *und* Verzeichnisse aus. Wenn also z.B. die Verzeichnisstruktur eines Datenträgers ermittelt werden soll, muß die Suchmaske `*.*` lauten, weil `scan_dir` sonst nicht alle Verzeichnisse erfaßt.

`attr` grenzt die Suche über die Attribute ein (Tab. 4.5). Es werden alle normalen Dateien gefunden plus die, die mindestens eines der Attribute tragen, die in `attr` spezifiziert wurden. Ein Ausnahme macht das `volume label`-Flag: Ist dieses gesetzt, wird nur der Datenträgername gefunden, der sich im Wurzelverzeichnis befindet. `step`

gibt die maximale Rekursionstiefe an. 0 bedeutet keine Rekursion, 255 bewirkt, daß der Dateibaum vollständig durchlaufen wird.

Bit	Bezeichnung	Bedeutung
0	`read-only`	nur-lesen
1	`hidden`	versteckt
2	`system`	Systemdatei (versteckt)
3	`volume label`	Datenträgername
4	`directory`	Verzeichnis
5	`archive`	Archivierungs-Flag
6 – 7		reserviert

Tabelle 4.5: Aufbau Attribut-Byte

Aus `path` und `mask` wird in `f_spec` zunächst die vollständige Suchmaske aufgebaut.

```
void scan_dir (int step, int attr, char path[], char mask[],
  void (*proc_path) (char path[], struct T_DTA *dta),
  void (*proc_file) (char path[], struct T_DTA *dta))
{ struct T_DTA dta;
  char f_spec[65], tmp[65];

  /* vollst. Dateispezifikation aufbauen                          */
  strcpy (f_spec, path);
  strcat (f_spec, mask);
```

Die Durchsuchung des angegebenen Verzeichnisses erfolgt mit den "C"-Funktionen `xfindfirst` und `xfindnext`, solange Einträge gefunden werden. Handelt es sich um ein Verzeichnis, werden die Sondernamen "." (gleiches Verzeichnis) und ".." (Vaterverzeichnis) ausgefiltert. Bei einem normalen Unterverzeichnis wird zur weiteren Verarbeitung die Anwenderfunktion `*proc_path` (Process Path) aufgerufen. Falls die max. Rekursionstiefe noch nicht erreicht ist, ruft sich `scan_dir` mit dem gefundenen Verzeichnis selbst auf.

Handelt es sich um eine Datei, erfolgt ein Aufruf der Funktion `*proc_file` (Process File). Durch die Übergabe von Zeigern auf Funktionen zur Verzeichnis- bzw. Dateiverarbeitung wird `scan_dir` allgemein verwendbar, kann aber trotzdem elegant für ganz spezielle Anwendungen eingesetzt werden.

```
  if (xfindfirst (f_spec, &dta, attr) == 0)
  { do
    { if (dta.attr & 0x10)              /* Verzeichnis?           */
      { if (dta.name[0] != '.')         /* Sonderverz. aussortieren*/
        { (*proc_path) (path, &dta);
          if (step)                     /* Rekursion?             */
          { sprintf  (tmp, "%s%s\\", path, dta.name);
            scan_dir (step - 1,attr,tmp,mask,proc_path,proc_file);
          };
        };
      }
      else                              /* Datei!                 */
      { (*proc_file) (path, &dta);
```

Funktion "Scan Directory":
`void scan_dir (int step, int attr, char path[], char mask[],`
`void (*proc_path) (char path[], struct T_DTA *dta),`
`void (*proc_file) (char path[], struct T_DTA *dta))`

Aufrufparameter:

`step`	max. Rekursionstiefe (0: keine Rekursion)
`attr`	Maske für Dateiattribute
`path`	Startpfad (mit Backslash beendet)
`mask`	Maske für Dateinamen
`proc_path`	Zeiger auf Funktion zur Pfadbearbeitung
`proc_file`	Zeiger auf Funktion zur Dateibearbeitung

Seiteneffekte:
Das aktuelle DTA wird verändert

Rückgabewert:
keiner

Tabelle 4.6: `scan_dir`: Dateisystem rekursiv durchsuchen

```
      };
    } while (xfindnext (&dta) == 0);
  };
};
```

Hinweise zu scan_dir. Würde vom Betriebssystem in der DTA-Struktur nicht die aktuelle Suchposition festgehalten, wäre Rekursion in der durchgeführten Weise nicht möglich. Bei der Rückkehr aus einem tiefer gelegenen Verzeichnis begänne die Suche von Neuem — eine Endlosschleife wäre das Resultat. Wie sähe eine alternative Implementation aus? Bei der Durchsuchung eines Verzeichnisses werden alle angetroffenen Verzeichniseinträge zunächst in einem lokalen Array von Strings gesammelt. Nach der Abarbeitung der Dateieinträge geht es an die Bearbeitung der Verzeichnisse, deren Abfolge durch einen lokalen Eintragszähler geregelt wird. Nach der Rückkehr von jeder Rekursion wird der Zähler erhöht und bei Erreichen des Endes der Verzeichnisliste zur aufrufenden Funktion zurückgekehrt. Durch das lokale Array und die Rekursion ist diese Variante allerdings viel speicheraufwendiger als die erste Version.

Doch nun zu `ChkSys`. Die Funktion `main` besteht aus drei Teilen. Zuerst wird die Liste der gefährlichen Dateien mit `load_list` in die durch `list_anchor` referenzierte, sequentiell vorwärts verkettete und sortierte Liste eingelesen.

```
/* einfuegen in "Typedefs"                                          */
struct T_ENTRY
{ char           *text;                /* Zeiger auf Text         */
  struct T_ENTRY *next;                /* Adresse naechster Knoten*/
};

/* einfuegen in "globale und externe Variablen"                     */
struct T_ENTRY *sub_anchor, *list_anchor;
```

```
int main (int argc, char *argv[])
{ struct T_ENTRY *ptr;                  /* Zeiger auf Dateieintrag */
  char  tmp[65];                        /* temp. String            */
  char  pos;                            /* Index Ende von TMP      */
  char  nr;                             /* Index 1. Argument       */

  /* zu wenig Parameter?                                           */
  if ((nr = get_arg (argc, argv, 0)) == 0)
  { fprintf (stderr,
      "Syntax: CHKSYS <start directory>\n");
    return (1);
  };

  /* Lese "schwarze Liste" ein                                     */
  add_path (argv[0], "chksys.lst", tmp);
  if (load_list (tmp, &list_anchor))
  { fprintf (stderr, "Couldn't read list file\n");
    return  (2);
  };
```

Der nächste Part liest die Verzeichnisstruktur, d.h. die Namen aller Verzeichnisse, in die Liste `sub_anchor` ein. Die Suche startet ab dem in der Kommandozeile angegebenen Verzeichnis und umfaßt alle Verzeichnisse des logischen Laufwerks. Das Startverzeichnis muß der Liste separat hinzugefügt werden, da es bei der Suche nicht mehr auftaucht. Da keine Dateinamen bearbeitet werden, füllt die "tue nichts"-Funktion `dummy` den offenen Platz im Aufruf von `scan_dir`.

```
  /* Lese und pruefe die Namen aller Verzeichnisse                 */
  printf ("scanning directory structure...\n");
  sub_anchor = NULL;
  strcpy (tmp, argv[nr]);
  add2list (&sub_anchor, tmp);          /* Startverz. hinzufuegen  */
  /* alle Verzeichnisse, "hidden" und "system" inklusive           */
  scan_dir (255, 0x16, tmp, "*.*", chk_subdir, dummy);
```

Die Funktion zur Verzeichnisbearbeitung `chk_subdir` setzt die Größe des Verzeichnisses auf 1, damit es zu keinen Beschwerden bei der Längenüberprüfung durch `check` kommt. Der Aufruf von `add2list` fügt das Verzeichnis zur Liste `sub_anchor` hinzu.

```
void chk_subdir (char path[], struct T_DTA *dta)
{ char tmp[132];

  dta -> size = 0x000000011;            /* Setze Groesse = 1       */
  check    ("", dta);
  sprintf  (tmp, "%s%s", path, dta -> name);
  add2list (&sub_anchor, tmp);
  return;
};
```

Der dritte Teil von `main` überprüft den für uns wesentlichen Inhalt der gefundenen Verzeichnisse, die Dateinamen. Dazu wird die Liste der Verzeichnisse `sub_anchor` durchlaufen und für jedes Verzeichnis `scan_dir` aufgerufen. Falls es sich nicht um das Wurzelverzeichnis handelt, wird der Pfadname um einen Backslash ergänzt. Die Bearbeitungsfunktion für die Dateinamen `chk_file` nimmt die eigentliche Überprüfung

vor. Verzeichnisse werden nicht bearbeitet, was durch die Verwendung von `dummy` zum Ausdruck kommt.

```
/* Lese und pruefe den Inhalt der Verzeichnisse                */
printf ("scanning files...\n");
ptr = sub_anchor;                     /* Start mit 1.Verzeichnis */
while (ptr != NULL)                   /* fuer alle Unterverz.    */
{ printf ("%s\n", ptr -> text);
  strcpy (tmp, ptr -> text);          /* fuege ggf. Backslash an */
  pos = strlen (tmp) - 1;
  if ((tmp[pos] != ':') && (tmp[pos] != '\\'))
  { strcat (tmp, "\\");
  };
```

```
    scan_dir (0, 0x16, tmp, "*.*", dummy, chk_file);
    ptr = ptr -> next;
  };
  delete_list (&sub_anchor);
  delete_list (&list_anchor);
  return (0);
};
```

Wie `chk_subdir` ruft `chk_file` die zentrale Funktion `check` auf, um den Dateinamen überprüfen zu lassen. Die von `check` aufgerufenen Funktionen `conv_date`, `conv_time`, `conv_attr` und `conv_size` wandeln die Einträge für Datum, Zeit, Attributbyte und Dateigröße in Strings um. Gleichzeitig wird geprüft, ob Datum oder Zeit unzulässig sind, das `hidden`- oder `system`-Attribut gesetzt oder die Größe 0 ist. Ein Rückgabewert von 0 bedeutet "keine Beanstandung".

`check` sammelt je nach Ergebnis der Konvertierungsroutinen die Warnmeldungen im String `warning`. Kam es zu mindestens einer Beanstandung des untersuchten Namens, wird eine Warnmeldung in die Reportdatei (Standardausgabe) geschrieben.

```
int check (char path[], struct T_DTA *dta)
{ char  warning[200];                   /* Warnmeldung             */
  char  full[65];                       /* voller Name (PATH&NAME) */
  char  date[11], time[9],              /* Dateiattribute (Strings)*/
        attr[9], size [8];
  int   flag;                           /* Fehlerflag              */
  BYTE  *p;                             /* Zeiger in DTA.NAME      */

  flag       = 0;
  warning[0] = '\0';
  strcpy (full, path);                  /* vollen Namen aufbauen   */
  strcat (full, dta -> name);

  if (conv_date (dta -> date, date))    /* illegales Datum?        */
  { strcat (warning, "invalid date ");
    flag = 0x01;
  };
>> FLAG: b0: conv_date, b1:conv_time, b2:conv_attr, b3:conv_size   <<
  if (selectp (list_anchor,dta -> name))/* in schwarzer Liste?     */
  { strcat (warning, "marked file ");
    flag |= 0x10;
  };
  p = (BYTE *)dta -> name;              /* Suche nach Steuerzeichen*/
  while (*p && isprint (*p))
  { p++;
  };
  if (*p)
  { strcat (warning, "control characters in name ");
    flag |= 0x20;
  };

  if (flag)                             /* Beanstandung?           */
  { printf ("\"%s\" -> %s\n", full, warning);
  };
  return (flag);
};
```

Hinweis. In die "schwarze Liste" sollten alle MS-DOS-Kommandos und Programme aufgenommen werden, die für den bestimmungsgemäßen Betrieb nicht notwendig sind. Dazu zählen möglicherweise Kopier- und Umbenennungsprogramme (`xcopy`,

`replace` etc.), Tools zur Manipulation der internen Struktur der Festplatte und Werkzeuge zur Programmerstellung. Ein Blick in die Aufstellung der von Computerviren benötigten Funktionen gibt evtl. weitere Hinweise auf kritische Programme.

4.3.2 Seal und Check Seal (chk_seal)

Problemstellung. Bei Dateien, die sich auf einem ungeschützten System befinden oder die versandt werden, ergibt sich ein Problem: Wie erkennt man unbefugte Veränderungen? Bei Daten, die sich auf dem eigenen System befinden, besteht wenigstens die Möglichkeit, Sicherheitskopien anzulegen und ggf. das Original mit der Arbeitskopie zu vergleichen. Für den Versand aber muß der Nachweis erbracht werden, daß die Datei von einem bestimmten Sender stammt (Authentifizierung) und nicht verändert wurde (Wahrung der Integrität).

Manuellen Kontrollverfahren haftet der Mangel an, daß der Anwender aktiv und vor jedem Programmstart die Prüfung durchführen muß. Wird versehentlich ein manipuliertes Programm gestartet, kann z.B. ein Virenscanner zum Infektionsverbreiter erster Kategorie werden. Man stelle sich nur den Fall vor, daß ein "Infect On Open"-Virus den Scanner verseucht hat.

Aufgabenbeschreibung. Es sind ein Programm `Seal` und eine Funktion `chk_seal` zu entwickeln, die eine Prüfsumme über bestimmte Daten bilden bzw. ein Programm automatisch beim Start überprüfen. Der Prüfsummenalgorithmus ist frei wählbar; auf der Programmdiskette wird das CRC-Verfahren verwendet.

Für die Authentifizierung und den Schutz der Integrität ist es notwendig, daß kein Dritter eine andere, zulässige Prüfsumme generieren kann. Aus diesem Grund verschlüsselt üblicherweise der Sender mit seinem geheimen Schlüssel (asymmetrisches Verfahren) die der Nachricht beigefügte Prüfsumme. In unserem Fall kann die Authentifizierung z.B. dadurch erfolgen, daß Generatorpolynom, Algorithmus, Startparameter und Prüfsumme auf einem anderen Weg als die Datei übermittelt werden (Brief, Telefon etc.). In diesem Fall übernehmen die Unterschrift bzw. die Stimme und gegenseitig abgefragtes Wissen die Rolle des Identifikators.

Systemarchitektur. `Seal` besteht aus vier Sektionen:

- `main` führt den Dialog mit dem Benutzer und ruft die einzelnen Funktionen auf
- `seal_disk` berechnet die Signatur für einen Datenträger
- `seal_boot` berücksichtigt nur die Urladeinformation (PBRs und MBRs, falls vorhanden)
- `seal_file` versiegelt eine einzelne Datei

Abbildung 4.2 stellt den Informationsfluß schematisch dar.

Funktionsbeschreibung. `main` bietet dem Anwender ein kurzes Menü an, nimmt die Auswahl entgegen und fragt spezifische Parameter ab (Dateiname, Laufwerk

Abbildung 4.2: Ein-/Ausgabe `Seal`

etc.). Dazu kommt noch die Initialisierung der CRC-Routinen mit `make_crc_table`. Jede Prüfsummenfunktion liefert die Signatur für das betreffende Objekt zurück.

seal_file. Fangen wir mit der einfachsten Funktion an. Der Aufruf von `seal_file` wird direkt an die Funktion `sig_crc` weitergeleitet, welche die Prüfsumme über eine Datei berechnet. Im Fehlerfall wird eine Nachricht ausgegeben, der Rückgabewert ist 0000_{16}.

seal_boot. Der Aufbau von `seal_boot` ist dem des Programms `ReadPart` sehr ähnlich. `main` findet seine Entsprechung in `seal_boot`, die rekursive Funktion `read_part_data` ist praktisch identisch mit `chk_part_data`. Der Unterschied besteht darin, daß die Textausgaben wegfallen und nach dem Einlesen eines MBRs oder PBRs mit `block_crc` die Prüfsumme über den Puffer berechnet wird. Neu ist auch der Parameter `*sig`, über den der Startwert vorgegeben und der berechnete Wert zurückgegeben wird.

```
/* einfuegen in "Defines"                                    */
#define CRC_START       0x0000          /* Startwert CRC-Funktionen*/
#define POLYNOMIAL      0xA001          /* Generatorpolynom        */

WORD seal_boot (int drive)
{ WORD sig;                             /* Signatur                */
  BYTE ldrive;                          /* logische Laufwerksnummer*/

  sig    = CRC_START;
  ldrive = 0;
  /* so lange der MBR gelesen werden kann                         */
  while (chk_part_data (&ldrive, (BYTE)drive, 0, 0, 1, &sig) == 0)
  { drive++;                            /* naechstes phys. Laufwerk*/
  };
  return (sig);
};
```

seal_disk. Wirklich neu und etwas aufwendiger ist die Funktion `seal_disk`. Das Gros der Routine macht Code zur Geschwindigkeitsoptimierung und Fehlerbehandlung aus. Zunächst ist festzustellen, wie viele Sektoren `sec_togo` insgesamt zu überprüfen sind. Diese Zahl ergibt sich aus dem Produkt von Clustern pro Datenträger `cpd` und Sektoren pro Cluster `spc`. Die Funktion $1C_{16}$ des Kernels `get_drive_data` ("C"-Funktion `drivedata`) liefert die nötigen Informationen (Tab. A.2).

```
/* einfuegen in "Defines"                                    */
```

```
#define BUF_SIZE        20480           /* Groesse CRC Puffer      */
```

```
/* einfuegen in "globale und externe Variablen"                    */
BYTE buf[BUF_SIZE];
WORD seal_disk (int drive)
{ struct T_BOOTSEC *pbr;                /* PBR                     */
  WORD bufs;                            /* Puffergroesse [Sektoren]*/
  WORD bps;                             /* Bytes pro Sektor (dummy)*/
  WORD cpd;                             /* Cluster pro Laufwerk    */
  BYTE spc;                             /* Sektoren pro Cluster    */
  BYTE far *media;                      /* Zeiger auf Media Byte   */
  WORD sec;                             /* aktueller Sektor        */
  WORD sec_togo;                        /* noch zu bearb. Sektoren */
  WORD sec_toread;                      /* zu lesende Sektoren     */
  WORD sig;                             /* Signatur                */
  WORD step;                            /* Sektoren pro Durchlauf  */
  WORD wait;                            /* Wartezaehler            */

  /* initialisiere Variablen, berechne Gesamtzahl Sektoren         */
  if (drivedata (1 + drive, &spc, &media, &bps, &cpd))
  { printf ("Invalid drive letter\n");
    return (0);
  };
  sec_togo = cpd * spc;
```

Die erzielte Lesegeschwindigkeit ist stark von der Anzahl der Sektoren abhängig, die mit einem Aufruf von `absread` gelesen werden. Tabelle 4.3 zeigt die Prüfzeit für eine 360kB-Diskette in Abhängigkeit von den gelesenen Sektoren pro Zugriff an. Starke Einbrüche sind bei 9 und 18 Sektoren erkennbar, was dem Umfang eines halben bzw. ganzen Zylinders entspricht (eine bzw. zwei Spuren). Weitere Vielfache von 9 bringen keinen Geschwindigkeitsgewinn, sondern fordern nur unnötig viel Pufferspeicher.

Abbildung 4.3: Abhängigkeit Prüfzeit von gelesenen Sektoren pro Zugriff

Ziel der Optimierung der Puffergröße ist es, im Rahmen des verfügbaren Speichers möglichst viele ganze Spuren gleichzeitig einzulesen. Der in `buf` eingelesene und über

`*pbr` ansprechbare PBR gibt Auskunft über die Anzahl der Sektoren pro Spur `spt` und Anzahl der Spuren pro Zylinder `heads` (= Anzahl Leseköpfe). `bufs` ist die Puffergröße in Sektoren zu 512 Bytes und wird mit 0 initialisiert. Die Anzahl der Puffersektoren wird so lange um die Anzahl der Sektoren pro Spur erhöht, bis entweder der komplette Zylinder abgedeckt ist (`heads` ist 0) oder die verfügbare Puffergröße überschritten würde.

```
/* berechne optimale Puffergroesse                                    */
xabsread  (drive, 1, 0, (void *)buf);
pbr      = (struct T_BOOTSEC *)buf;
bufs     = 0;
while ((bufs + pbr -> spt < (BUF_SIZE >> 9)) && (pbr -> heads)--)
{ bufs += pbr -> spt;
};
```

`step` enthält die Anzahl der Sektoren pro Lesezugriff und wird mit der Puffergröße `bufs` vorbelegt. Solange noch Sektoren zur Bearbeitung anstehen, wird versucht, `step` oder die restliche Anzahl Sektoren zu lesen. Im Erfolgsfall wird die Prüfsumme berechnet und die Anzahl der verbleibenden Sektoren `sec_togo` reduziert.

```
step     = bufs;
sec      = 0;
wait     = 0;
sig      = CRC_START;
/* bearbeite Diskette                                                 */
while (sec_togo)
{ sec_toread = min (sec_togo, step);
  if (xabsread (drive, sec_toread, sec, (void *)&buf) == 0)
  { sig = block_crc (buf, sec_toread << 9, sig);
    sec_togo -= sec_toread;
    sec      += sec_toread;
  }
  else                                 /* Fehler!                     */
```

Bei einem Lesefehler, mit dem praktisch auf jeder Festplatte zu rechnen ist, wird versucht, das Sektorenbündel in einzelnen Sektoren zu lesen (`step` = 1).

Ein Fehler, der bei Schrittweite 1 auftritt, ist nicht behebbar. In diesem Fall wird eine kurze Meldung ausgegeben und der defekte Sektor übersprungen. Nachdem das fehlerhafte Sektorbündel in Einzelschritten gelesen wurde (`wait` wird von `step` auf 0 heruntergezählt), erhält `step` seinen alten Wert `bufs` zurück.

```
  { if (step != 1)                     /* Start Fehlerbehandlung  */
    { wait = bufs;                     /* nur 1 Sektor/Zugriff    */
      step = 1;
    }
    else                               /* erneuter Fehler?        */
    { printf ("Read error at sector %u!\n", sec);
      sec++;                           /* lasse Sektor aus        */
      sec_togo--;
    };
  };
  if (wait == 0)                       /* Ende Fehlerbehandlung   */
  { step = bufs;                       /* max. Sektoren/Zugriff   */
  }
  else
  { wait--;
  };
};
```

```
  return (sig);
};
```

chk_seal. Programme können sich beim Start selbst überprüfen, indem sie über die eigene Datei eine Prüfsumme bilden und mit einer gespeicherten vergleichen. Die Referenzdaten sind zweckmäßigerweise Bestandteil der Datei und nicht extern abgespeichert. Auf diese Weise wird der Versand von Programmen möglich, die im Falle einer Manipulation auf dem Transportweg den Anwender vor sich selbst warnen. Eine Reihe von Antivirusprogrammen wie `ViruScan` von McAfee machen von dieser Methode Gebrauch.

Die Funktion `check_seal`, die unmittelbar nach dem Programmstart aufgerufen werden sollte, vergleicht die aktuelle Prüfsumme mit einem in der Programmdatei enthaltenen Siegel. Der Rückgabewert macht eine Aussage darüber, ob das Programm verändert wurde oder nicht.

Funktion. Eine Funktion von `AVConfig` berechnet die Prüfsumme über das fertig kompilierte Programm und hängt diese an die Datei an. Dieses *Programmsiegel* hat die Form

```
struct T_SEAL
{ WORD id;
  WORD sig;
};
```

Zur Überprüfung der Integrität wird "`in`" auf das dem Programm angehängte Siegel positioniert, dieser Stand des Dateizeigers als Programmlänge in `f_size` festgehalten und das Siegel in `seal` eingelesen.

```
/* einfuegen in "Defines"                                        */
#define BUF_SIZE        512
#define CRC_START       0x0000
#define POLYNOMIAL      0xA001

int check_seal (char *argv[])
{ BYTE buf[BUF_SIZE];                   /* CRC-Puffer             */
  FILE *in;                             /* Eingabedatei           */
  struct T_SEAL seal;                   /* Programmsiegel         */
  long to_go;                           /* Anzahl zu lesende Bytes */
  size_t bytes;                         /* Anzahl gelesene Bytes  */
  WORD sig;                             /* aktuelles Siegel       */

  if ((in = fopen (argv[0], "rb")) == NULL)
  { printf ("CHECK_SEAL: Couldn't open program file\n");
    return (-1);
  };

  /* lese und pruefe Programmsiegel                                */
  fseek (in, -(long)sizeof (struct T_SEAL), SEEK_END);
  to_go = ftell (in);
  if (fread (&seal, sizeof (struct T_SEAL), 1, in) != 1)
  { fclose (in);
    printf ("CHECK_SEAL: Couldn't read program seal\n");
    return (-2);
  };
```

Die Identifikationsnummer `id` hat normalerweise den Wert 2342_{16}. Dadurch läßt sich bestimmen, ob ein Programm ein gültiges Siegel trägt oder nicht.

```
  if (seal.id != 0x2342)
  { fclose (in);
    printf ("CHECK_SEAL: Seal not found\n");
    return (-3);
  };
```

Zur Berechnung der aktuellen Prüfsumme kann nicht die Funktion `sig_crc` verwendet werden, welche die Prüfsumme über die *gesamte* Programmdatei *inklusive Siegel* bilden würde. Die eigene Prüfroutine läßt die letzten Bytes, die das Siegel enthalten, weg. Diese sind nämlich erst nach Berechnung des Programmsiegels durch `AVConfig` hinzugekommen und würden das Ergebnis verfälschen. Rückgabewert ist das Ergebnis des Tests auf Ungleichheit zwischen der Referenzprüfsumme `seal.sig` und der zuvor berechneten aktuellen Prüfsumme `sig`.

```
  /* berechne aktuelles Siegel                                    */
  sig = CRC_START;
  make_crc_table (POLYNOMIAL);
  rewind (in);
  while ((bytes = fread (buf, 1, min (to_go, BUF_SIZE), in)) != 0)
  { sig = block_crc (buf, bytes, sig);
    to_go -= bytes;
  };
  fclose (in);

  return (sig != seal.sig);
};
```

Funktion "Check Programm Seal" `check_seal`**:**
`int chk_seal (char *argv[])`

Aufrufparameter:

`argv`	Array mit Zeigern auf die Kommandozeilenparameter

Seiteneffekte:
keine

Rückgabewert:
0: Siegel intakt; 1: Siegel beschädigt; -1: Fehler

Tabelle 4.7: `check_seal`: Überprüfe Programmsiegel

4.3.3 Check State (ChkState)

Problemstellung. Wie unter 4.3.1 "Check System" bereits angesprochen, nehmen manche Softwareanomalien Manipulationen am Dateibestand und am Dateisystem vor. Ebenso können Anwender unbefugt Dateien ins System einspielen oder aus dem System entfernen. `ChkSys` kann aber nicht zwischen regulären und hinzugekommenen Dateien unterscheiden, solange sie nicht durch besondere Attribute auffallen. Gelöschte Dateien und Verzeichnisse werden überhaupt nicht erkannt, weil kein Vergleich mit einem früheren Zustand des Systems vorgesehen ist.

Drei Dinge weisen auf bereits erfolgte illegale Aktivitäten im System hin, falls kein Befugter die Operationen vorgenommen hat:

1. Neue Dateien unbekannter Herkunft (Einschleusung).
2. Veränderte Dateien (Inhalt und Attribute wie Datum, Zeit etc.; Infektion, Manipulation).
3. Gelöschte Dateien (Schadensfunktion).

Die mithin wichtigste Eigenschaft einer Datei ist natürlich ihr Inhalt, und der wird von MS-DOS weder geschützt noch überwacht. Softwareanomalien, allen voran die Viren, verändern aber oft die gespeicherte Information.

Veränderungen an Dateiattributen (analog: Verzeichnisattribute) werden ebenfalls nicht überwacht. Der Begriff "Attribute" sei hier etwas weiter gefaßt als sonst und bezeichne alle Eigenschaften einer Datei. Durch den Befehl `dir` werden vier Merkmale angezeigt: Name und Größe sowie Datum und Zeit der letzten Bearbeitung mit Schreibzugriff. Weiterhin verwaltet MS-DOS noch ein Attributbyte und einen Verweis auf den ersten Cluster der Datei. Von den vier Attributen im Sinne von MS-DOS läßt sich nur das `readonly`-Flag über den Befehl `attrib` direkt beeinflussen. Das `archive`-Flag wird automatisch nach jeder Schreiboperation gesetzt; `hidden`- und `system`-Attribut sind überhaupt nicht zugänglich.

Aufgabenbeschreibung. Zu entwickeln ist das Programm `ChkState`, das folgende Aufgaben hat:

- Eine Liste mit der Verzeichnisstruktur und dem Dateibestand eines Datenträgers erstellen (Zustand des Dateisystems festhalten; quantitativ).
- Für jedes Unterverzeichnis und jede Datei Datum, Zeit, Attribut, Größe und eine Signatur speichern (Zustand der Objekte des Dateisystems festhalten; qualitativ).
- Den aktuellen Status des Dateisystems mit einem früheren vergleichen und dabei einen Report mit gelöschten, hinzugekommenen und veränderten Verzeichnissen und Dateien erstellen (Änderungsreport, quantitativ und qualitativ).

Bei der Auswahl des Signaturverfahrens müssen wir Sicherheit und Geschwindigkeit gegeneinander abwägen. Prüfsummenverfahren wie simples Aufaddieren und CRC-Verfahren sind zwar relativ schnell, aber auch leicht zu durchbrechen. Viel sicherer sind kryptische Prüfsummen, wie sie z.B. das DES- oder MD4RSA-Verfahren erzeugen. Dieser Vorteil wird mit einem erheblich größeren Zeitaufwand erkauft. Für ein Virus gilt allerdings, daß dieses weder das Signaturverfahren noch die Stelle kennt, an der die Vergleichsdaten gespeichert sind.

`AVWatchI` ist ein noch zu entwickelnder Watcher, der ähnlich dem Betriebssystem S3 von Cohen die Integrität von Programmen vor dem Start überprüft. Dazu benötigt `AVWatchI` eine Referenzliste, die sicherheitsrelevante Daten wie z.B. die Prüfsumme über das Programm enthält. Damit das TSR-Programm klein bleibt und schnell auf die Referenzdaten zugreifen kann, ist der Aufbau der Referenzliste entsprechend zu gestalten. Allgemein gilt für den Prüfalgorithmus hinsichtlich des Einsatzes in `AVWatchI`:

- Die Prüfsumme muß schnell zu berechnen sein.
- Der Prüfsummenalgorithmus sollte nicht viel Platz verbrauchen.
- Der Zugriff auf Daten der Referenzliste muß schnell und einfach erfolgen können.

Somit sind schnelle Implementationen des CRC-Algorithmus das Mittel unserer Wahl.

Entwurf der Datenstrukturen und Algorithmen. Die einfachste Methode zur Erzeugung der Vergleichsliste besteht darin, mit `scan_dir` das Dateisystem zu durchforsten und jede Datei inklusive Verzeichnis abzuspeichern. Das brächte eine Menge Nachteile mit sich. Durch das Abspeichern der Verzeichnisse bei *jedem* Dateinamen wird viel Speicherplatz verbraucht. Für jede vorgefundene Datei muß die alte Liste nach einem passenden Eintrag durchsucht werden, der bei zufälliger Verteilung der Namen nach durchschnittlich $\frac{n}{2}$ Suchschritten gefunden würde, wenn n die Länge der Liste ist. Hinzugekommene Verzeichnisse und Dateien würden dadurch entdeckt, daß in der alten Liste kein passender Eintrag existiert. Wie steht es mit gelöschten Objekten? Diese wären in der alten Liste überzählig und müßten z.B. durch Markierung aller bereits angesprochenen Einträge ausgefiltert werden.

Das alles ist vom softwareästhetischen Standpunkt her gesehen nicht sehr schön. Wie bei vielen anderen Gelegenheiten werden die gestellten Aufgaben durch die Verwendung sortierter Listen vereinfacht, obwohl der Sortiervorgang selbst wieder Zeit verbraucht. Betrachten wir zunächst den Vergleich des Dateibestands und -zustands innerhalb eines Verzeichnisses. Im Beispiel wurde die Datei "`b`" gelöscht, "`e`" kam hinzu (Tab. 4.8). Der aktuelle Eintrag in der alten Liste wird jeweils mit dem in der neuen verglichen. Dabei gibt es drei mögliche Ergebnisse (Tab. 4.9):

1. Beide Einträge sind gleich → keine Änderung im Bestand, Attribute vergleichen, beide Zeiger erhöhen.
2. Der neue Eintrag ist größer → die durch den alten Eintrag bezeichnete Datei wurde gelöscht, zum nächsten Eintrag in alter Liste übergehen.
3. Der neue Eintrag ist kleiner → die durch den neuen Eintrag spezifizierte Datei kam hinzu, zum nächsten Eintrag in neuer Liste wechseln.

alte Liste	neue Liste
a	a
b	c
c	d
d	e
f	f

Tabelle 4.8: Beispiel "Dateibestand"

Dazu kommen noch Sonderfälle, die dadurch entstehen, daß eine Liste bis an ihr Ende gelesen wurde, die andere aber noch Einträge enthält. Zusammenfassend gesagt:

alte Liste	neue Liste	Vergleich	Bedeutung	Aktion
a	a	a = a	o.k.	beide Listen weiter
b	c	b < c	b gelöscht	alte Liste weiter
c	c	c = c	o.k.	beide Listen weiter
d	d	d = d	o.k.	beide Listen weiter
f	e	f > e	e neu	neue Liste weiter
f	f	f = f	o.k.	beide Listen weiter
Ende	Ende		Ende!	

Tabelle 4.9: Beispiel "Vergleich von Dateibeständen"

Es wird die Liste ausgewählt, die nicht leer ist und deren aktueller Eintrag kleiner als der der anderen ist. Ist eine Liste leer, wird stets aus der anderen gelesen. Sind beide Listen aufgebraucht, ist der Vergleich beendet.

Genauso gehen wir beim Bestandsvergleich auf Verzeichnisebene vor. Existiert das Unterverzeichnis, wird zur nächsten Hierarchiestufe gewechselt und der Inhalt, d.h. der Dateibestand, wie oben besprochen überprüft. Wurde ein Verzeichnis gelöscht oder hinzugefügt, sind alle Dateien darin ebenfalls gelöscht oder neu. Um den Programmieraufwand gering zu halten, wird der Vergleichsroutine für den Dateibestand in diesem Fall vorgetäuscht, daß eine der beiden Listen leer ist.

Zur internen Verwaltung von Verzeichnis- und Dateidaten werden sequentielle, vorwärts verkettete sortierte Listen verwendet. Weil das Paket zur Listenverwaltung nur mit konventionellen Strings nach der "C"-Konvention arbeitet, müssen die Daten in Textform vorliegen. Ursache dafür ist, daß Daten im DTA, z.B. die Uhrzeit, Bytes mit dem Wert 0 enthalten könnten, was gleichzeitig "String-Ende" bedeutet. Außerdem lassen sich die Dateieigenschaften in Textform einfacher vergleichen (Abb. 4.4).

Abbildung 4.4: Aufbau Textform Datei-/Verzeichniseintrag bei `ChkState`

Die externe Referenzdatei (ebenfalls sortiert) ist zweistufig in Verzeichnisliste und Dateiliste organisiert, um anderen Programmen den schnellen und einfachen Zugriff zu ermöglichen. Jeder Verzeichniseintrag (Format `struct T_SUBDIR`) enthält Angaben über den ersten und letzten zugehörigen Dateieintrag (Format `struct T_FILE`) in der Dateiliste (s.a. "`get_ref`" in A.4). Alle Datensätze haben eine feste Länge und Aufteilung, damit der Datensatz und einzelne Felder direkt adressierbar sind.

```
struct T_SUBDIR                         /* Verzeichniseintrag (75B)*/
{ char  name[65];                       /* Verzeichnisname         */
  WORD  start;                          /* erster Dateieintrag     */
  WORD  end;                            /* letzter Dateieintrag    */
  WORD  date;                           /* Dateiattribute          */
  WORD  time;
  WORD  attr;
};

struct T_FILE                           /* Dateieintrag (39 Bytes) */
{ char  name[13];                       /* Dateiname               */
  WORD  date;                           /* Dateiattribute          */
  WORD  time;
  DWORD size;
  WORD  attr;
  DWORD sign[4];                        /* Signatur                */
};
```

Konvertierungsfunktionen. Um bei der Speicherung der externen Referenzliste Platz zu sparen, sind in den Strukturen `T_FILE` und `T_SUBDIR` die Angaben für Zeit, Datum, Attribute und Größe in kodierter Form abgelegt (identisch zur Kodierung im DTA). Die Funktionen `dta2date`, `dta2time`, `dta2attr` und `dta2size` sind Bestandteil der Bibliothek `avsys.lib` und wandeln die Einträge für Datum, Zeit, Attribute und Größe im DTA-Format in Strings um. Für die umgekehrte Richtung sind die Funktionen `date2dta`, `time2dta`, `attr2dta` und `size2dta` zuständig. Wer sich die Implementierung der Konvertierungsfunktionen, die wegen ihres einfachen Aufbaus nicht besprochen werden, ersparen möchte, kann die Referenzdaten natürlich auch direkt im Textformat abspeichern.

In Richtung DTA → Text ist der erste Parameter bei den Konvertierungsfunktionen stets ein `WORD` (Ausnahme: `DWORD` bei `SIZE`), der zweite ein String. In Richtung Text → DTA läuft die Parameterübergabe gerade umgekehrt. `file2text`, `text2file`, `subdir2text` und `text2subdir` bauen auf den genannten Funktionen auf und wandeln den Inhalt der Strukturen `T_FILE` bzw. `T_SUBDIR` in die zugehörige Textform und zurück um. Abb. 4.4 zeigt anschaulich, wie die Textform der Datei- und Verzeichniseinträge aufgebaut ist.

Systemarchitektur. Da es wieder um die Durchsuchung von Dateibeständen geht, kann auf die Funktion `scan_dir` zurückgegriffen werden. Wie bei `ChkSys` ist die Suche in die zwei Ebenen Verzeichnisstruktur und Dateibestand aufgegliedert, die durch die Funktionen `scan_structure` und `scan_files` realisiert werden. Auf beiden Ebenen kommt der oben beschriebene Vergleichsalgorithmus zur Anwendung.

Funktionsbeschreibung. `ChkState` erwartet als Aufrufparameter eine Liste von Laufwerksangaben, z.B. "`c: d:`". `main` initialisiert mit Hilfe von `init` das Programm, ruft für jeden Parameter der Kommandozeile `scan_structure` auf und nimmt die Nachbereitung vor.

```
/* Defines                                                      */
#define POLYNOMIAL      0xA001
#define CRC_START       0
#define BUF_SIZE        (18 * 512)      /* 18 Sektoren          */
```

```
/* Globale Variablen                                              */
FILE    *in_subdir;                    /* Verzeichnisliste Eingabe*/
FILE    *in_file;                      /* Dateiliste Eingabe      */
FILE    *out_subdir;                   /* Verzeichnisliste Ausgabe*/
FILE    *out_file;                     /* Dateiliste Ausgabe      */
struct T_ENTRY *sub_anchor;            /* Verzeichnisliste        */
struct T_ENTRY *file_anchor;           /* Dateiliste              */
struct T_ENTRY *list_anchor;           /* Auswahlliste            */
int     sout_file, eout_file;          /* 1., letzter Eintrag     */
size_t  buf_size;                      /* Puffergroesse von BUF   */
BYTE    *buf;                          /* Puffer fuer CRC-Funkt.  */

int main (int argc, char *argv[])
{ int arg;

  if (init (argc, argv) < 0)           /* Fehler bei INIT?        */
  { return (1);
  };

  for (arg = 1; arg < argc; arg++)     /* fuer alle Laufwerke     */
  { scan_structure (argv[arg]);
  };
  delete_list (&list_anchor);
  fclose (in_subdir);
  fclose (in_file);
  fclose (out_subdir);
  fclose (out_file);
  return (0);
};
```

`init` beginnt mit der Initialisierung der Variablen `sout_file` und `eout_file`, deren Funktion unter "Lesen/Schreiben der Referenzdaten" erläutert wird. Nächster Punkt ist das Öffnen aller Referenzdateien (s.a. Abb. 4.5). `chkstate.sui` und `chkstate.fii`[6] enthalten die Verzeichnis- bzw. Dateidaten der letzten Überprüfung. Falls eine der beiden Dateien nicht existiert, wird statt dessen die `nul`-Datei geöffnet und so eine leere Datei simuliert.

Das Ergebnis der aktuellen Überprüfung wird in den Dateien `chkstate.suo` und `chkstate.fio` gespeichert, die entweder neu angelegt oder überschrieben werden. Vor dem nächsten Start von `ChkState` sind die Ausgabedateien in die Eingabedateien umzubenennen (z.B. mit "`copy chkstate.??o *.??i`"). In `chkstate.lst` kann der Anwender analog zu `ChkSys` Dateien spezifizieren, die bei der Kontrolle berücksichtigt und in die Referenzliste aufgenommen werden sollen (Empfohlen: alle ausführbare Dateien). Zuletzt werden noch die CRC-Funktionen initialisiert und Pufferspeicher reserviert.

```
int init (int argc, char *argv[])
{ char f_spec[65];

  sout_file = 0;
  eout_file = 0;

>> oeffne alle Dateien; falls eine Referenzdatei fehlt,           <<
>> oeffne statt dessen "NUL"-Datei (falls Eingabe)                <<
>> IN_SUBDIR ("chkstate.sui"); OUT_SUBDIR ("chkstate.suo")        <<
>> IN_FILE   ("chkstate.fii"); OUT_FILE   ("chkstate.fio")        <<
```

[6] **Su**bdirectory/**Fi**le **i**nput

```
  /* lade Auswahlliste                                          */
  add_path (argv[0], "chkstate.lst", f_spec);
  if (load_list (f_spec, &list_anchor))
  { fprintf (stderr, "Couldn't load selection-list\n");
    return  (-6);
  };
  make_crc_table (POLYNOMIAL);
  /* reserviere Pufferspeicher fuer CRC-Berechnung              */
  if ((buf = (BYTE *)malloc (BUF_SIZE)) == NULL)
  { return (-7);
  };

  return (0);
};
```

Abbildung 4.5: Ein-/Ausgabe `ChkState`

Lesen der aktuellen Daten. Wir werden uns zunächst mit einigen Supportfunktionen beschäftigen, um die Beschreibung des Hauptteils übersichtlich zu halten. Die Routinen `add_subdir` und `add_file` werden von der Funktion `scan_dir` aufgerufen. Beide fügen den Namen des Verzeichnisses bzw. der Datei und den sonstigen Inhalt des DTA in Textform in die entsprechende Liste ein. Da die Parameterliste der Bearbeitungsfunktionen durch die Deklaration von `scan_dir` bereits vorgegeben ist, sind die Listen-Anker global definiert; eine Methode, die sonst nicht zu empfehlen ist.

Zwischen der Bearbeitung eines Verzeichnisses und einer Datei bestehen geringfügige Unterschiede. Verzeichnisse haben weder eine Größe noch eine Signatur; deshalb werden diese Felder mit 0 bzw. einer Folge von '-' (Minus) Zeichen vorbelegt.

```
/* Defines                                                      */
#define EXT_STOP        0x01
#define INT_STOP        0x02

void add_subdir (char path[], struct T_DTA *dta)
{ char  tmp[160];                       /* Hilfsstring        */
  char  name[65];                       /* Verzeichnisname    */
  char  date[11];                       /* Dateiattribute     */
  char  time[9];
```

```
  char  attr[17];
  char  size[8];

  /* DTA-Eintraege in Textstrings umwandeln                          */
  sprintf (name, "%s%s\\", path, dta -> name);
  dta2date (dta -> date, date);
  dta2time (dta -> time, time);
  dta2attr (dta -> attr, attr);
  dta2size (dta -> size, size);
  /* baue Textstring zusammen                                        */
  sprintf   (tmp, "%-64s %s %s %s       "
                  "0 --------------------------------",
             name, date, time, attr, size);
  add2list  (&sub_anchor, tmp);         /* Eintrag -> in Liste     */
  return;
};
```

Bei Dateien kommt zu den Daten aus dem DTA die zu berechnende Signatur hinzu. Das Verfahren kann durch den Anwender frei gewählt werden. Für die Signatur sind max. 32 Zeichen entsprechend 16 Bytes in sedezimaler Darstellung vorgesehen, wie sie z.B. das MD4RSA-Verfahren liefert.

```
void add_file (char path[], struct T_DTA *dta)
{ char  tmp[100];                     /* Hilfsstring             */
  char  date[11];                     /* Dateiattribute          */
  char  time[9];
  char  attr[17];
  char  size[8];
  char  f_spec[65];                   /* vollst. Dateiname       */
  WORD  sig;                          /* Pruefsumme              */

  /* DTA-Eintraege in Textstrings umwandeln                        */
  if (selectp (list_anchor, dta -> name))/* Datei in Auswahlliste? */
  {
>> Konvertieren von DATE, TIME, SIZE, ATTR (s.a. ADD_SUBDIR)       <<
    sprintf  (f_spec, "%s%s", path, dta -> name);
    sig = CRC_START;
    if (sig_crc (buf, BUF_SIZE, f_spec, &sig) < 0)
    { fprintf (stderr, "Computing of signature failed\n");
      getch ();
    };
    sprintf (tmp, "%-12s %s %s %s %s "
                  "00000000000000000000000000000%04.4X",
             dta -> name, date, time, attr, size, sig);
    add2list (&file_anchor, tmp);       /* Eintrag -> in Liste     */
  };
  return;
};
```

Lesen/Schreiben der Referenzdaten. Eine wichtige Rolle spielen die vier Zähler `sin_file` (Startindex Eingabe von `chkstate.fii`), `ein_file` (Endindex Eingabe), `sout_file` (Startindex Ausgabe auf `chkstate.fio`) und `eout_file` (Endindex Ausgabe). Abbildung 4.6 dürfte den im folgenden Text geschilderten Sachverhalt etwas erhellen.

`read_subdir` liest den nächsten Eintrag aus der Verzeichnisliste (Eingabe) und setzt dabei `sin_file` und `ein_file`, die den zugehörigen Bereich in der Dateiliste (Eingabe) markieren. Die Routine liefert den Wert 1 zurück, falls das physikalische oder

logische Ende der Liste erreicht wurde; sonst den Wert 0. Unter "logischem Ende" wird der trennende Eintrag in der Verzeichnisliste vor den Daten des nächsten Laufwerks verstanden. `sin_file` bezeichnet zugleich die Position des nächsten zu lesenden Dateieintrags und wird bei jedem Aufruf von `read_file` automatisch erhöht. Die Funktion gibt einen von 0 verschiedenen Wert zurück, falls das Ende des Teilbereichs in der Dateiliste überschritten wurde (`*sin_file > ein_file`).

`write_file` schreibt einen Dateieintrag an die durch `eout_file` bezeichnete Stelle der Dateiliste (Ausgabe) und erhöht den Zähler. `write_subdir` schließlich überträgt `sout_file` und `eout_file` in den aktuellen Verzeichniseintrag und fügt diesen an die Verzeichnisliste (Ausgabe) an. Zusätzlich erhält `sout_file` den Wert von `eout_file`.

Abbildung 4.6: Dateiindizes und deren Verwendung bei `ChkState`

Verzeichnisebene. Nun kommen wir zu den Hauptfunktionen. `scan_structure` liest zunächst die komplette Dateistruktur, d.h. die Namen der Verzeichnisse, in die mit `sub_anchor` referenzierte interne Liste ein. Das Wurzelverzeichnis wird dabei separat behandelt, weil es nicht als Verzeichnis an sich bei der Suche auftaucht.

```
void scan_structure (char drive[])
{ struct T_ENTRY *ptr;                  /* Zeiger in interne Liste */
  struct T_SUBDIR subdir;               /* Verzeichniseintrag      */
  int    decide;                        /* Entscheidungsflag       */
  int    sin_file, ein_file;            /* Start/Ende externe Liste*/
  char   ext[200]                       /* externer Eintrag, Text  */
  char   hlp[65];                       /* Hilfsstring             */
  char   stop;                          /* Stop-Flag               */

  /* lese Verzeichnisse ein                                        */
```

```
printf ("scanning directory structure...\n");
sub_anchor = NULL;
/* Eintrag fuer Wurzelverzeichnis generieren, in Liste eintragen */
sprintf (subdir.name, "%c:\\", tolower (drive[0]));
subdir.date = 0x0000;
subdir.time = 0x0000;
subdir.attr = 0x0000;
subdir2text (&subdir, ext);
add2list  (&sub_anchor, ext);
get_name  (ext, hlp);                /* Start mit Wurzelverz.   */
scan_dir  (255, 0x16, hlp, "*.*", add_subdir, dummy);
```

Als Vergleichsliste dient der extern gespeicherte alte Bestand. `stop` zeigt in bitweiser Kodierung an, ob die interne Liste mit den aktuellen Daten oder die externe Liste mit den Referenzdaten das Ende erreicht hat. Eine `AND`-Operation mit `INT_STOP` bzw. `EXT_STOP` filtert das entsprechende Bit heraus.

Zunächst werden die ersten beiden Elemente der Listen eingelesen. Die `while`-Schleife wird so lange durchlaufen, bis beide Listen das Ende erreicht haben. Je nach Vergleichsergebnis `decide` werden entsprechende Informationen in die neue Bestandsliste und die Reportdatei (Standardausgabe) geschrieben sowie das nächste Element der internen und/oder externen Liste gelesen. In jedem Fall übernimmt die Funktion `scan_files` die weitere Bearbeitung eines gelöschten, existierenden oder hinzugekommenen Verzeichnisses.

```
/* bearbeite Verzeichnisse                                          */
printf ("scanning files...\n");
ptr  = sub_anchor;                   /* Start mit 1. Element    */
stop = (ptr == NULL) * INT_STOP;
stop |= read_subdir (&sin_file, &ein_file, ext);

while (stop != (INT_STOP | EXT_STOP))
{ decide = stricmpu (ptr -> text, ext, ' ');

  /* Verzeichnis hinzugefuegt                                       */
  if (((decide < 0) || (stop & EXT_STOP)) && !(stop & INT_STOP))
  { printf ("directory \"%s\" added\n",
      get_name (ptr -> text, hlp));
    scan_subdir (EXT_STOP, ptr -> text, sin_file, ein_file,
                 sout_file, &eout_file);
    write_subdir (&sout_file, eout_file, ptr -> text);
    ptr  = ptr -> next;
    stop |= (ptr == NULL) * INT_STOP;
    continue;
  };

  /* Verzeichnis geloescht                                          */
  if (((decide > 0) || (stop & INT_STOP)) && !(stop & EXT_STOP))
  { printf ("directory \"%s\" deleted\n",
      get_name (ext, hlp));
    scan_subdir (INT_STOP, ptr -> text, sin_file, ein_file,
                 sout_file, &eout_file);
    stop |= read_subdir (&sin_file, &ein_file, ext);
    continue;
  };

  /* Verzeichnis existiert                                          */
  if (decide == 0)
```

```
    { printf ("%s\n", get_name (ptr -> text, hlp));
      check_stamp (ptr -> text, ext, 65);
      scan_subdir (0x00, ptr -> text, sin_file, ein_file,
                   sout_file, &eout_file);
      write_subdir (&sout_file, eout_file, ptr -> text);
      ptr  = ptr -> next;
      stop |= (ptr == NULL) * INT_STOP;
      stop |= read_subdir (&sin_file, &ein_file, ext);
      continue;
    };
  };
```

Nach Verlassen des Schleifenkörpers wird an die Verzeichnisliste noch ein Leereintrag angehängt, um die Daten verschiedener Laufwerke zu trennen.

```
  /* schreibe Trenneintrag                                          */
  write_subdir (&sout_file, eout_file, "");
  delete_list (&sub_anchor);
  return;
};
```

Dateiebene. `scan_subdir` ist der kompliziertere Part des Funktionsduos, weil hier noch drei Modi unterschieden werden. Die Variable `stop` enthält wie bei `scan_structure` Informationen darüber, welche Liste *nicht* bearbeitet werden darf. Nur kann hier durch `mode` ein Startwert vorgegeben und dadurch von vornherein eine Liste ausgeschlossen werden. Das macht Sinn, wenn ein neues Verzeichnis hinzugekommen ist (es existiert keine interne Liste) oder ein bestehendes gelöscht wurde (es existiert keine externe Liste).

Eingangs der Funktion liest `scan_dir` die Dateidaten in die Liste mit dem Anker `file_anchor` ein, falls `mode` dies zuläßt.

```
void scan_subdir (int mode, char path[], int sin_file, int ein_file,
                  int sout_file, int *eout_file)
{ struct T_ENTRY *ptr;                  /* Zeiger auf int. Eintrag */
  char  hlp[65];                        /* Hilfsstring             */
  char  ext[200];                       /* externer Eintrag        */
  int   decide;                         /* Entscheidungsflag       */
  int   stop;                           /* Stop-Flag               */

  stop = mode;
  *eout_file = sout_file;
  if (!(mode & INT_STOP))               /* int. Liste bearbeiten?  */
  { file_anchor = NULL;                 /* Ja: lese Dateien ein    */
    get_name (path, hlp);
    scan_dir (0, 0x16, hlp, "*.*", dummy, add_file);
    ptr = file_anchor;
    stop |= (ptr == NULL) * INT_STOP;
  };
  if (!(mode & EXT_STOP))               /* ext. Liste bearbeiten?  */
  { stop |= read_file (&sin_file, ein_file, ext);
  };
```

Die weitere Verarbeitung erfolgt analog zu `scan_structure`, wobei die Endekriterien wegen der Organisation der Listen anders formuliert sind. Die interne Liste ist zu Ende, wenn der letzte Folgezeiger auf kein weiteres Element verweist, sondern `NULL` ist (Bit `INT_STOP` in `stop` gesetzt). Das Ende der externen Liste ist durch das Ergebnis

von `read_file` bestimmt (Bit `EXT_STOP` in `stop` gesetzt). Die Funktion schließt mit der Freigabe der Dateiliste `file_anchor` ab.

```
  while (stop != (INT_STOP | EXT_STOP))
  { decide = stricmpu (ptr -> text, ext, ' ');

    /* Datei geloescht (externe Liste lesen)                    */
    if (((decide > 0) || (stop & INT_STOP)) && !(stop & EXT_STOP))
    { printf ("file \"%s\" deleted\n", get_name (ext, hlp));
      stop |= read_file (&sin_file, ein_file, ext);
      continue;
    };

    /* Datei hinzugefuegt (interne Liste lesen)                 */
    if (((decide < 0) || (stop & EXT_STOP)) && !(stop & INT_STOP))
    { printf ("file \"%s\" added\n", get_name (ptr -> text, hlp));
      write_file (eout_file, ptr -> text);
      ptr = ptr -> next;
      stop |= (ptr == NULL) * INT_STOP;
      continue;
    };

    /* Datei existiert                                          */
    if (decide == 0)
    { check_stamp (ptr -> text, ext, 13);
      write_file (eout_file, ptr -> text);
      ptr = ptr -> next;
      stop |= (ptr == NULL) * INT_STOP;
      stop |= read_file (&sin_file, ein_file, ext);
      continue;
    };
  };
  delete_list (&file_anchor);
  return;
};
```

Überprüfung der Attribute. `check_stamp` nimmt die Überprüfung vor, wenn `scan_structure` oder `scan_subdir` zwei gleiche Einträge finden. `stamp1` und `stamp2` dienen zur Vereinfachung des Zugriffs auf die Datenfelder der in Textform vorliegenden Einträge `entry1` und `entry2`. Da Verzeichnis- und Dateinamen verschieden lang sind, muß der Funktion über `offset` mitgeteilt werden, ab welcher Stelle des Eintrags die Attributfelder stehen.

```
/* einfuegen in "Typedefs"                                      */
struct T_FSTAMP                         /* Textformat Attribute    */
{ char  date[11];                       /* "jjjj-mm-tt "           */
  char  time[9];                        /* "hh:mm:ss "             */
  char  attr[17];                       /* "          ADVSHR "     */
  char  size[8];                        /* "nnnnnnn "              */
  char  sign[32];                       /* "ccc...ccc"             */
};

int check_stamp (char entry1[], char entry2[], char offset)
{ struct T_FSTAMP *stamp1, *stamp2;     /* zur Vereinfachung       */
  char hlp[65];                         /* temp. String            */
  char changes[80];                     /* "veraendert"-Meldung    */
  char flag;                            /* "veraendert"-Bitmuster  */

  /* vereinfache Zugriff auf Attributfelder                       */
```

```
  stamp1 = (struct T_FSTAMP *)(entry1 + offset);
  stamp2 = (struct T_FSTAMP *)(entry2 + offset);
  flag       = 0;
  changes[0] = '\0';
```

Die einzelnen Attributfelder werden getrennt auf Übereinstimmung verglichen. `strncmp` wird deshalb verwendet, weil die Felder Teile eines einzigen Strings sind und nicht durch 0-Bytes beendet werden. Im Fehlerfall wird ein entsprechender Text zur Warnmeldung `changes` hinzugefügt und das korrespondierende Bit im Fehlerflag `flag` gesetzt. Die Funktion gibt — falls ein Fehler aufgetreten ist — die gesammelten Warnmeldungen aus und zeigt die neuen und alten Attributwerte zur Begutachtung durch den Anwender an.

```
  /* vergleiche Datum                                              */
  if (strncmp (stamp1 -> date, stamp2 -> date, 10))
  { strcat (changes, "date ");
    flag = 0x01;
  };
>> dito Zeit (8 Bytes), Attribute (16), Groesse (7), Signatur (32) <<
  if (flag)                          /* Veraenderungen?         */
  { get_name (entry1, hlp);
    printf ("%.12s: %shas changed\nwas/is:\n", entry1, changes);
    printf ("%s\n%s\n", entry2 + 13, entry1 + 13);
  };
  return (flag);
};
```

Erläuterungen: get_name extrahiert aus einem Texteintrag den Namen der Datei oder des Verzeichnisses. Dieser ist mit Leerzeichen aufgefüllt und von anderen Angaben durch ein Leerzeichen getrennt. `get_name` verwendet die Funktion `strcpyu`, die einen String bis zu einem Trennzeichen, in diesem Fall ein Leerzeichen, kopiert.

Erweiterungsmöglichkeiten. `ChkState` ist eine Plattform für alle Verfahren, die auf Vergleichsdaten zurückgreifen müssen. Über die beschriebenen Maßnahmen hinaus können noch weitere Prüfroutinen installiert werden. Der Zustand einer Datei ist z.B. nicht nur durch die oben angeführten Attribute bestimmt, sondern durch weitere, nicht so augenscheinliche Eigenschaften:

- Erster Cluster der Datei (Start des Speicherplatzes, welcher der Datei zugeordnet ist),
- Abfolge der Cluster ("Cluster Chain"; Lage der Datei auf dem Datenträger).
- Position (Startadresse) des Dateieintrags im Verzeichnis.
- Inhalt der unbenutzten Bereiche im Dateieintrag.

Manipulationen auf Dateiebene verändern u.U. die angeführten Eigenschaften. Eine gelöschte und später wieder auf den Datenträger kopierte Datei kommt, falls inzwischen andere Dateioperationen erfolgt sind, höchstwahrscheinlich an einem anderen Platz in Verzeichnis und Datenbereich zu liegen. Die konventionelle Überprüfung könnte keinen Unterschied zum vorherigen Zustand feststellen, wohl aber ein Test der erweiterten Attribute. Diese bei Manipulationen beizubehalten ist als sehr schwer wenn nicht

unmöglich einzustufen. Noch ein Hinweis: Programme zur Defragmentierung von Dateibeständen verändern die eben angeführten Attribute ebenfalls. Nach jedem Reorganisationslauf müßte die erweiterte Version von `ChkState` den Zustand des Dateisystems neu erfassen.

4.4 Kommandos mit Kontrollfunktionen

4.4.1 AVCopy

Problemstellung. Die im Kapitel 2 entwickelte Methode der kontrollierten Isolation läßt sich nicht allein mit Watchern verwirklichen. Bei z.B. einem Kopiervorgang werden eine Vielzahl von Dateien aus den unterschiedlichsten Gründen geöffnet und geschlossen. In erster Linie werden Daten gelesen und geschrieben, aber auch die Existenz von Pfaden und Dateien überprüft sowie Dateidatum und -zeit ermittelt. Der Zusammenhang zwischen den einzelnen Operationen ist nicht oder nur schwer nachvollziehbar. Ein externes Kommando wie `xcopy` hingegen wird für einen ganz bestimmten Zweck aufgerufen, nämlich zum Kopieren von Dateien. Eine im Kopierprogramm befindliche Kontrollfunktion ist genau über die Aufgabe jedes Dateizugriffs informiert. Insbesondere ist die Überwachung von Transportwegen möglich, da Quell- und Ziellaufwerk, -verzeichnis und -datei bekannt sind. Auch Typwechsel (ausführbar/nicht ausführbar) und das Umbenennen von Programmen unterliegen exakter Kontrolle.

Aus der Bestrebung, eine Transportkontrolle zu errichten, und aus den angeführten Gründen ergibt sich die Notwendigkeit, ein eigenes externes Kopierprogramm `AVCopy` und Umbenennungsprogramm `AVRename` zu erstellen (Abb. 4.7).

Abbildung 4.7: Ein-/Ausgabe `AVCopy/AVRename`

Aufgabenbeschreibung AVCopy/AVRename. Der Transport von Dateien impliziert die Existenz einer Quelle und eines Ziels. Die Ortsangabe soll auf (logische)

Laufwerke beschränkt sein. Höchstens auf Festplatten wäre evtl. die weitere Gliederung in Unterverzeichnisse sinnvoll. Doch sind z.B. in Rechenzentren die Laufwerke meist so organisiert, daß sich auf einer Partition das Betriebssystem und alle für den Betrieb notwendigen Programme befinden, und auf einer anderen Partition die Anwender zu Hause sind. Das Laufwerk mit der Betriebssoftware (Compiler, Linker, Textverarbeitung etc.) sollte vor Aktivitäten der Benutzer sicher sein, während sich auf der Anwenderpartition beliebige Programme befinden dürfen.

In Zusammenarbeit mit einem Watcher lassen sich wirkungsvolle Schutzmaßnahmen realisieren. Ein Beispiel: Der Start von Programmen auf Floppylaufwerken ist verboten, ebenso das Kopieren von Programmen von Floppy auf Festplatte. Damit können Viren nur noch als Quelltext eingeschleppt werden — von "Versehen" oder "Ungeschick" kann dann nicht mehr die Rede sein. Auch das Tarnen und Enttarnen von Programmen wird unmöglich, falls Typwechsel verboten sind.

Typwechsel. Wir unterscheiden primär zwei Dateitypen: Daten (nicht ausführbar) und Programme (ausführbar). "Ausführbar" sei hier definiert als "kann direkt über die Funktion "Load and Execute" des Kernel gestartet werden". Zur Beschreibung des scheinbaren und des tatsächlichen Dateityps werden folgende Begriffe, wie auch "Virus", der Biologie entlehnt:

Phänotyp: Erscheinungsbild, -form eines Organismus. Hier: Der Name, insbesondere die Erweiterung (Endung) einer Datei, die anschaulich über den Typ Auskunft gibt (z.B. `com`, `doc`, `pas`).

Genotyp: Gesamtheit der Erbfaktoren eines Lebewesens. Hier: Der tatsächliche Inhalt einer Datei.

Der Phänotyp und der Genotyp einer Datei müssen nicht notwendigerweise identisch sein. Die Datei `tunix.dat` kann ein Virusprogramm enthalten, `h.exe` vom Inhalt her ein gewöhnlicher Text sein. Während der zweite Fall ungefährlich ist, kann mit Hilfe der ersten Methode eine scheinbar harmlose Datei auf Festplatte gebracht werden, ohne daß ein Kontrollprogramm dies erkennt. Das Umbenennen und/oder Kodieren einer Datei wird im weiteren als *tarnen* bzw. *enttarnen* bezeichnet. Damit ein getarntes Programm wieder ausführbar wird, muß es der Benutzer noch in `*.exe` oder `*.com` umbenennen und evtl. dekodieren. Weil MS-DOS dem Anwender die Vergabe von Dateinamen nicht vorschreibt, eignen sie sich, wenn überhaupt, nur begrenzt zur Identifizierung des Inhalts. Es ist deshalb ein Verfahren zu entwickeln, um Programm- und sonstige Dateien durch Analyse des Inhalts voneinander zu unterscheiden.

Namensänderungen von Programmen sind für die Sicherheit interessant, weil bei Watchern Programmnamen oft mit Rechten verknüpft sind. Durch einfaches Umbenennen wird u.U. aus einem beliebigen Programm ein privilegiertes mit Rechten zum Schreiben von ausführbaren Dateien. Handelt es sich um einen Virus oder einen Trojaner, ist der Tag schnell verdorben. Unter UNIX sind Namensänderungen kein Problem, weil die Rechte direkt mit den Dateieinträgen verknüpft sind. Der Dateiname ist nur ein Feld in diesem Eintrag und kann beliebig geändert werden. Dateien gleichen Namens im gleichen Verzeichnis können also auch unterschiedliche Rechte haben.

Der Funktionsumfang von `copy` wird stark zusammengestrichen, damit die Hauptlast der Programmierung nicht auf luxuriösen Kopieroptionen, sondern auf den Kontrollfunktionen liegt. Es kann und muß genau eine Quell- und Zielspezifikation angegeben werden. Die Verknüpfung von Dateien per '+'-Zeichen ist nicht möglich; die Schalter für Kopieren im ASCII- ("`/A`") und Binärmodus ("`/B`") werden nicht erkannt.

Das Rechtekonzept. Befassen wir uns zuerst mit den Rechten, die einem Transportweg verliehen werden können. Die Fett gedruckten Buchstaben bezeichnen die Abkürzungen, die in der Rechtedatei `t_rights.lst` (**T**ransport Rights) Verwendung finden werden (Kleinschreibung!). `*_TR` sind symbolische Konstanten, mit denen sich per `and`-Operation das entsprechende Bit aus dem Rechtewort isolieren läßt.

- Umbenennen von Dateien (Change of **N**ame `TR_NAMECHG`)
- Verschieben von Dateien (Change of **P**ath `TR_PATHCHG`)
- Unterschied zwischen Phäno- und Genotyp (Different **T**ypes `TR_TYPEDIF`)
- Änderung Phänotyp (**C**hange of Type `TR_TYPECHG`)
- Kopieren von Programmen (Transport of **E**xecutables `TR_EXEC`)
- Kopieren von Daten (Transport of **D**ata `TR_DATA`)

Ein Rechteeintrag (eine Zeile) in `t_rights.lst` hat die Form

```
Recht     ::= <Laufwerk> <Rechte> <Laufwerk>
Laufwerk  ::= <Buchstabe>:
Buchstabe ::= A|B...Y|Z
Rechte    ::= {Option(en)}
Option    ::= n|p|t|c|e|d (Kleinbuchstaben!)
```

Intern werden die Rechte in dem zweidimensionalen Array `t_rights` gespeichert. Die Nummern des Quell- und des Ziellaufwerks (`A:` = 0) dienen als Index. So findet sich z.B. das Rechtewort für den Transportweg `A:`→`B:` an der Stelle `t_rights[0][1]`.

Systemarchitektur. `AVCopy` und `AVRename` haben einen aus vier Schichten bestehenden hierarchischen Aufbau (Abb. 4.8; für `AVRename` "`copy`" durch "`rename`" ersetzen):

- `main` prüft die Eingaben des Anwenders auf Vollständigkeit, reserviert Pufferspeicher und liest die Rechtedatei ein,
- `gen_copy` normalisiert und ergänzt Quell- und Zielpfade und generiert aufgrund der Dateispezifikationen Dateinamen,
- `chk_copy` überprüft die angeforderte Operation anhand der Rechte auf Zulässigkeit,
- `phys_copy` wählt den korrekten Lese-/Schreibmodus aus und kopiert die Datei.

Normalerweise werden allen Dateien im Binärmodus kopiert, d.h. Steuerzeichen in der Information werden ignoriert. Besonders zu beachten ist die Behandlung von Gerätedateien, aus denen *nicht* im Binärmodus gelesen werden darf. Andernfalls würde

Abbildung 4.8: Systemarchitektur `AVCopy/AVRename`

die Markierung "Ende der Übertragung" (EOT; **E**nd **O**f **T**ransmission) nicht erkannt und die Leseroutine würde sich in einer Endlosschleife erhängen. Die Funktion `isatty` ("is a TTY" = "ist ein serielles Gerät[7]") bestimmt für eine geöffnete Datei, ob es sich wirklich um eine Datei oder um ein Gerät handelt.

Funktionsbeschreibung. Die Hauptfunktion `main` ist für einleitende Maßnahmen wie Kontrolle der Aufrufparameter und Reservierung von Pufferspeicher zuständig.

```
/* Defines                                                          */
/* Puffergroesse = 18 Sektoren mit 512 Bytes                        */
#define BUF_SIZE        (18 * 512)
#define M_DRIVE         10

/* globale Variablen                                                */
WORD    t_rights[M_DRIVE][M_DRIVE];     /* Rechte-Tabelle          */
BYTE    *buf;                           /* Zeiger auf Kopier-Puffer*/

int main (int argc, char *argv[])
{ char *err_msg[] =                     /* Fehlermeldungen GEN_COPY*/
  { "",
    "Illegal source path",
    "Illegal destination path"
  };
  char  f_name[65];                     /* vollst. Name Rechtedatei*/
  int   source, dest;                   /* Nummer Quell-/Zielangabe*/
  char  err;                            /* Fehlernummer            */
```

[7] engl. **t**ele**ty**pewriter = Fernschreiber

```
  /* pruefe Argumente, reserviere Speicher, lade Transportrechte   */
  if ((source = get_arg (argc, argv, 0)) == 0)
  { fprintf (stderr, "Must provide source argument -> abort\n");
    return (1);
  };
  if ((dest = get_arg (argc, argv, 1)) == 0)
  { fprintf (stderr, "Must provide destination argument -> abort\n");
    return (1);
  };
  if ((buf = (BYTE *)malloc (BUF_SIZE)) == NULL)
  { fprintf (stderr, "Couldn't allocate memory -> abort\n");
    return (2);
  };
  add_path (argv[0], "t_rights.lst", f_name);
  if (read_t_rights (f_name, t_rights))
  { fprintf (stderr, "Couldn't read rights file\n");
    return  (3);
  };

  err = -gen_copy (argv[source], argv[dest]);
  fprintf (stderr, "%s\n", err_msg[err]);
  return (0);
};
```

gen_copy. Die Quell- und Zielangabe in `r_source` bzw. `r_dest` werden nach dem gleichen Verfahren bearbeitet. Dateiangaben können sich auf das aktuelle Laufwerk und das aktuelle Verzeichnis beziehen. `AVCopy` muß aber in der Lage sein, das Laufwerk sicher zu erkennen. Deshalb werden Dateipfade zuerst "normalisiert", d.h. in eine Standardform gebracht. Diese beinhaltet Laufwerksangabe, Pfad ab Wurzelverzeichnis und den Dateinamen. `norm_path` erkennt zusätzlich, ob die übergebene Dateispezifikation ein gültiges Verzeichnis ist. In diesem Fall kann `AVCopy` die Angabe automatisch um `*.*` (alle Dateien des Verzeichnisses) ergänzen. Im Fall eines unzulässigen Pfades wird die Ausführung abgebrochen.

```
int gen_copy (char r_source[], char r_dest[])
{ char  *err_msg[] =                    /* Fehlermeldungen         */
  { "",
    "Couldn't open source file",
    "Couldn't read source timestamp",
    "Couldn't open destination file",
    "Couldn't write destination file",
    "Couldn't write destination timestamp",
    "Destination and source are same",
    "Transport between subdirectories not allowed",
    "Renaming files not allowed",
    "Phenotype and genotype of destination would be different",
    "Type changing not allowed",
    "Transport of code not allowed",
    "Transport of data not allowed"
  };
  struct T_DTA dta;
  char  source[65], dest[65];           /* Quell-/Zieldatei        */
  char  source_path[65], dest_path[65]; /* Quell-/Zielpfad         */
  char  dest_name[13];                  /* Zielmaske               */
  char  fill[65];                       /* Zielname                */
  char  err;                            /* Fehlernummer            */
```

```
  strcpy (source, r_source);
  switch (norm_path (source))
  { case NR_PATH:
      comp_fspec (source, "*.*");
      break;
    case NR_INVAL:
      return (-2);
  };
>> analog fuer dest, r_dest; return-Wert -3                    <<
```

Eigentlich wäre es ausreichend, nur das Laufwerk zu ermitteln, denn die angestrebten Schutzmaßnahmen sind alle laufwerksbezogen. Es ist aber möglich, als zukünftige Erweiterung einen pfadorientierten, differenzierteren Schutz zu realisieren. Watcher, die sich bei der Kontrolle von Aufrufen am Dateinamen orientieren, benötigen ebenfalls eine vollständige Normalisierung. Doch zurück zu `gen_copy`.

Quell- und Zielangabe werden in ihre Bestandteile Pfad und Dateinamen zerlegt. Für jede gefundene Datei läuft die gleiche Prozedur ab. Die Quelldatei `source` wird aus dem Quellpfad `source_path` und dem gefundenem Namen `dta.name` zusammengesetzt. Der Name der Zieldatei ergibt sich aus dem Zielpfad `dest_path` und dem Ergebnis der `fill_in`-Operation von `dta.name` in die Zielmaske `dest_name`. Falls die Überprüfung von Quelle und Ziel mit `chk_copy` zu keiner Beanstandung führt, erfolgt der eigentliche Kopiervorgang mit `phys_copy`.

```
  split_fspec (SM_PATH, source, source_path);
  split_fspec (SM_PATH, dest, dest_path);
  split_fspec (SM_NAME, dest, dest_name);
  printf (">avcopy \"%s\" -> \"%s\"\n", source, dest);

  if (xfindfirst (source, &dta, 0x00))  /* keine Datei gefunden?   */
  { return (-1);
  };

  do
  { strcpy (source, source_path);
    strcat (source, dta.name);
    strupr (source);
    fill_in (dta.name, dest_name, fill);
    strcpy (dest, dest_path);
    strcat (dest, fill);
    strupr (dest);
    printf ("Copy \"%s\" -> \"%s\"\n", source, dest);
    if ((err = -chk_copy (source, dest)) == 0)
    { err = -phys_copy (source, dest);
    };
    if (err)
    { fprintf (stderr, "%s\n", err_msg[err]);
    };
  } while (xfindnext (&dta) == 0);

  return (0);
};
```

chk_copy. Untersucht wird, ob der Kopiervorgang von `source` nach `dest` unter Berücksichtigung der Transportrechte `t_rights` zulässig ist. Sind Quelle und Ziel identisch, führt dies sofort zu einem Abbruch.

```
/* einfuegen in "Defines"; s. Text                                  */
#define TR_DATA         0x0001
#define TR_EXEC         0x0002
#define TR_TYPECHG      0x0004
#define TR_TYPEDIF      0x0008
#define TR_PATHCHG      0x0010
#define TR_NAMECHG      0x0020

int chk_copy (char source[], char dest[])
{ WORD  rights;
  char  source_part[65], dest_part[65]; /* Teil Quell-/Zielpfad    */
  char  source_geno, source_pheno;      /* Geno-/Phaenotyp Quelle  */
  char  dest_pheno;                     /* Phaenotyp Ziel          */

  /* Quelle und Ziel identisch?                                     */
  if (stricmp (source, dest) == 0)
  { return (-6);
  };
```

Die Funktion meldet einen Fehler zurück, falls ein oder mehrere der genannten Rechte verletzt werden. `rights` wird zur Vereinfachung mit dem für die Operation relevanten Rechtewort aus `t_rights` geladen.

```
  rights = t_rights[source[0] - 'A'][dest[0] - 'A'];
  split_fspec (SM_PATH, source, source_part);
  split_fspec (SM_PATH, dest,   dest_part);
  /* check "path change"                                            */
  if (stricmp (source_part, dest_part) && !(rights & TR_PATHCHG))
  { return (-7);
  };
  split_fspec (SM_NAME, source, source_part);
  split_fspec (SM_NAME, dest,   dest_part);
  /* check "name change"                                            */
  if (stricmp (source_part, dest_part) && !(rights & TR_NAMECHG))
  { return (-8);
  };
  /* check "type difference"                                        */
  source_pheno = get_phenotype (source);
  if ((source_geno = get_genotype (source)) < 0)
  { fprintf (stderr, "Couldn't determine genotype of sourcefile\n");
  };
  source_geno = (source_geno != 0);
  dest_pheno   = get_phenotype (dest);
  if ((dest_pheno != source_geno) && !(rights & TR_TYPEDIF))
  { return (-9);
  };
  /* check "type change"                                            */
  if ((source_pheno != dest_pheno) && !(rights & TR_TYPECHG))
  { return (-10);
  };
  /* check "transport executable"                                   */
  if (source_geno && !(rights & TR_EXEC))
  { return (-11);
  };
  /* check "transport data"                                         */
  if (!source_geno && !(rights & TR_DATA))
  { return (-12);
  };

  return (0);
};
```

Typüberprüfung (Namensanalyse). `get_phenotype` ermittelt den Phänotyp einer Datei, der durch die Erweiterung bestimmt ist (Tab. 4.10). Falls diese nicht existiert, wird die Funktion vorzeitig mit dem Rückgabewert 0 (kein Programm) beendet. Eine Einstufung als Programm erfolgt, wenn die Erweiterung in der Liste der Programmerweiterungen enthalten ist (Funktionswert 1). Diese ist im Quelltext festgelegt und sollte zumindest die Standarderweiterungen `exe`, `com`, `app`, `obj`, `lib` und `bat` enthalten.

Funktion "Get Phenotype":
`int get_phenotype (char f_spec[])`

Aufrufparameter:

`f_spec`	Dateiname

Seiteneffekte:
keine

Rückgabewert:
0: nicht ausführbar; 1: ausführbar

Tabelle 4.10: `get_phenotype`: Bestimme Phänotyp (einer Datei)

Typüberprüfung (Dateianalyse). Um den Schutz der Festplatte vor "getarnten" Programmen zu gewährleisten, genügt es nicht, nur auf die Erweiterung zu achten. Da diese nicht zwingend den Inhalt bezeichnet, benötigen wir andere Kriterien zur Bestimmung des Dateityps. Dateien lassen sich prinzipiell in zwei Kategorien einordnen: Solche mit fixen, überprüfbaren Attributen und solche, deren Inhalt analysiert werden muß. MS-DOS zum Beispiel verlangt, daß `exe`-Dateien mit den Buchstaben `MZ` in den ersten zwei Bytes markiert sind. Außer den Linkern, die sich an diese Konvention halten müssen, gibt es eine Reihe von Programmen, die die von ihnen erzeugten Dateien mit Schlüsselwörtern kennzeichnen (z.B. `pkpak` mit `PK` etc.).

Die zur zweiten Kategorie gehörenden `com`-Dateien hingegen haben keinen bestimmten Dateikopf und sind somit schlecht von z.B. Graphikdaten zu unterscheiden. Als einziger Hinweis kann der oft am Anfang des Programms vorhandene Sprung über Datenbereiche hinweg zum eigentlichen Programmcode dienen. Dieser ist allerdings nicht zwingend vorhanden oder könnte auch zufällig Bestandteil einer beliebigen Datei sein. Ebenso könnte eine Textdatei mit den Buchstaben `MZ` beginnen, was zwar unwahrscheinlich, aber möglich ist.

Wir benötigen also einen Algorithmus, der Textdateien von Programmdateien unterscheiden kann. Eine typische Eigenschaft von ASCII-Dateien sind längere Zeichenketten (Strings), die durch nur wenige Steuerzeichen unterbrochen werden. Das gilt eingeschränkt auch für Dateien, wie sie Textverarbeitungsprogramme beim Speichern von Dokumenten erstellen. Solche Strings sind zwar u.U. auch in Programmen enthalten (Eingabeaufforderungen, Hilfetexte etc.), aber nur zu einem erheblich geringeren Anteil als der eigentliche Programmcode.

Hier setzt die Analysefunktion `get_genotype` an (Tab. 4.11). Die ersten Bytes (max. 4096) der zu untersuchenden Datei werden in einen Puffer eingelesen. Ermittelt werden die Anzahl der Strings und der absolute Anteil am Text (in Bytes). Ein String ist hierbei definiert als eine mindestens zwei Zeichen lange Folge, deren Elemente ASCII-Zeichen im Wertebereich [9,127] sind. Aus dieser Information wird der relative Anteil der Strings an der Datei und die durchschnittliche Stringlänge berechnet. Eine Datei wird als Text angesehen, wenn

1. die durchschnittliche Stringlänge mindestens eine drittel Zeile (26 Zeichen) beträgt und
2. der Anteil von Strings an der Datei mindestens 90% ist.

Funktion "Get Genotype":
`int get_genotype (char f_spec[])`

Aufrufparameter:

`f_spec`	Dateiname

Seiteneffekte:
keine

Rückgabewert:
0: Text; 1: kein Text oder Datei ist Gerät (→ nicht überprüfbar)

Tabelle 4.11: `get_genotype`: Bestimme Genotyp (einer Datei)

Tests an verschiedenartigen Dateien haben ergeben, daß diese Parameter zu guten Ergebnissen führen. Insbesondere wurde kein Fall beobachtet, bei dem `get_genotype` ein Programm irrtümlich als Text identifiziert hätte. Text- und Nicht-Text-Dateien sind also leicht und relativ sicher zu unterscheiden. Die Sicherheit wird aber mit dem Nachteil erkauft, daß evtl. benötigte Daten nicht auf Festplatte kopiert werden können, da `AVCopy` z.B. Grafikdaten als "nicht Text" erkennt und deshalb zurückweist.

Die Unterscheidung zwischen Programmen und Nicht-Programmen ist nur schwer oder gar nicht möglich und mit großer Unsicherheit behaftet. Im Zweifelsfall wird man dem Ausschluß aller Nicht-Text-Dateien den Vorzug geben, weil dann mit Sicherheit nur Quelltexte kopiert werden können.

phys_copy. Das eigentliche Kopieren der Dateien ist eine triviale Aufgabe. Wichtig ist jedoch der Zugriff im korrekten Modus, der von dem mit `is_device` ermittelten Typ abhängt. Normale Dateien sind im Binärmodus zu kopieren, Gerätedateien dürfen nur im ASCII-Modus geöffnet werden. Als nützliches Extra sollte die Zieldatei die Zeitmarke der Quelldatei erhalten, weil sonst alle bearbeiteten Dateien Datum und Zeit des Kopiervorgangs aufweisen, was nicht immer erwünscht ist. Der folgende Code stellt eine Lösungsmöglichkeit dar. Hinweis: Die Option `S_IWRITE` bewirkt, daß die Datei nicht mit dem sonst üblichen `readonly`-Attribut angelegt wird.

```
int phys_copy (char source[], char dest[])
{ struct T_FTIME f_time;                /* Zeitmarke der Quelldatei*/
  size_t bytes;                         /* Anzahl gelesene Bytes   */
  int f_source, f_dest;                 /* Zeiger Quell-/Zieldatei */
  char err;                             /* Fehlercode              */

  /* Dateien im korrekten Modus oeffnen                            */
  if ((f_source = open (source, O_RDONLY |
    (is_device (source) ? O_TEXT : O_BINARY))) == -1)
  { return (-1);
  };
  /* Zeitmarke der Quelldatei merken                               */
  if (xgetftime (f_source, &f_time) == -1)
  { return (-2);
  };
  if ((f_dest = open (dest, O_CREAT | O_WRONLY |
    (is_device (dest) ? O_TEXT : O_BINARY), S_IWRITE)) == -1)
  { return (-3);
  };

  /* Datei kopieren                                                */
  err = 0;
  while ((bytes = read (f_source, buf, min (BUF_SIZE, 32767))) != 0)
  { if (write (f_dest, buf, bytes) == -1)
    { fprintf (stderr, "Write error\n");
      err = -4;
    };
  };

  /* Zeitmarke Zieldatei = Zeitmarke Quelldatei                    */
  if (xsetftime (f_dest, &f_time) == -1)
  { return (-5);
  };
  close (f_source);
  close (f_dest);

  return (err);
};
```

4.4.2 AVRename

AVRename. Bei der Entwicklung von `AVRename` zeigt sich, daß sich der modulare Aufbau von Programmen bezahlt macht. Die Architektur ist die gleiche wie bei `AVCopy`. Statt `*_copy` tragen die Funktionen die Endung `*_rename`. Lediglich die unterste Ebene, `phys_rename`, ist natürlich von `phys_copy` verschieden.

gen_rename. Der einzige Unterschied zu `gen_copy` besteht in der anderen Behandlung von Weglaßwerten. `AVRename` verfügt nämlich über mächtigere Fähigkeiten als das gewöhnliche `rename`. Dateien können nicht nur umbenannt, sondern zwischen Verzeichnissen eines Laufwerks verschoben werden. Beim Original-`rename` ist die Angabe eines Zielpfades weder erlaubt noch macht sie Sinn, denn es können nur Dateien innerhalb eines Verzeichnisses umbenannt werden. Deshalb verwendet `rename` standardmäßig den Quellpfad als Zielpfad. Damit die Kompatibilität gewahrt bleibt, arbeitet `AVRename`, wenn der Zielpfad nicht angegeben wird, auf die gleiche Weise.

Anstatt der ganzen Routine seien hier nur die Änderungen angegeben:

```
/* einfuegen in Variablendeklaration                                  */
  char  no_path;

/* einfuegen vor "switch (norm_path (dest))"                          */
  split_fspec (SM_PATH, dest, dest_path);
  no_path = (dest_path[0] == '\0');

/* einfuegen nach den drei "split_fspec"                              */
  if (no_path)                          /* kein Zielpfad?           */
  { strcpy (dest_path, source_path);    /* Zielpfad = Quellpfad     */
  };
  printf ("Rename \"%s\" -> \"%s\"\n", source, dest);

/* die while-Schleife erhaelt eine neue Abbruchbedingung              */
  } while ((xfindnext (&dta) == 0) && (err != 2));
```

phys_rename. Das Umbenennen von Dateien erledigt die Kernelfunktion 56_{16} "Rename File", die als "C"-Funktion `renfile` in `msdos_s.lib` zu Verfügung steht. Lediglich die Fehlerbehandlung verbraucht ein paar Zeilen Code, ansonsten ist der Aufwand mit `phys_copy` nicht zu vergleichen. Der Rückgabewert -2 signalisiert einen Fehler, der auch bei allen folgenden Dateien auftreten würde. `gen_copy` bricht daraufhin die Ausführung ab (s. neue Abbruchbedingung).

```
int phys_rename (char source[], char dest[])
{ int err;

  switch (renfile (source, dest))
  { case 0x00:                          /* kein Fehler              */
      err = 0;
      break;
    case 0x11:                          /* unterschiedl. Laufwerke */
      err = -2;
      break;
    case 0x02:                          /* Datei nicht gefunden     */
    case 0x05:                          /* Datei schreibgeschuetzt */
    default:
      err = -1;
  };
  return (err);
};
```

Erläuterungen: norm_path2 . Wie schon angesprochen, können Dateiangaben unvollständig sein. Darüber hinaus schaffen die reservierten Verzeichnisnamen "`.`" (gleiches Verzeichnis) und "`..`" (Vaterverzeichnis) Verwirrung. Welche Datei verbirgt sich z.B. hinter der Angabe "`.\..\..\.\ROBERTS`"? Lautet der aktuelle Pfad "`C:\JULIA\LOVES\ROMEO`", ist die richtige Antwort "`C:\JULIA\ROBERTS`". Ziel der Normalisierung ist also eine Dateispezifikation mit vollständigem Pfad ohne relative Verzeichnisangaben.

Funktion. Zuerst wird die Dateiangabe in die drei Teile Laufwerk `new`, Verzeichnis `subdir` und Name `name` aufgespalten. In `new` wird die normalisierte Dateiangabe beginnend mit dem Laufwerk aufgebaut. Falls dieses fehlt, wird das aktuelle Laufwerk verwendet.

```
void norm_path2 (char f_spec[])
{ char subdir[65];                      /* Unterverz. in F_SPEC    */
  char name[13];                        /* Name in F_SPEC          */
  char new[80];                         /* F_SPEC, normalisiert    */
  char *to, *scan, *sub;                /* Zeiger in Strings       */

  strupr (f_spec);
  split_fspec (SM_DRIVE, f_spec, new);
  split_fspec (SM_DIRECTORY, f_spec, subdir);
  split_fspec (SM_NAME, f_spec, name);

  if (new[0] == '\0')                   /* keine Laufwerksangabe?  */
  { new[0] = 'A' + xgetdisk ();
    new[1] = ':';
    new[2] = '\0';
  };
```

Ist der Pfad nicht ab Wurzelverzeichnis oder überhaupt nicht angegeben, fügt `norm_path2` der Laufwerksangabe in `new` das aktuelle Verzeichnis hinzu. Falls das aktuelle Verzeichnis nicht das Wurzelverzeichnis ist und `name` Zeichen enthält[8], kommt noch ein Backslash zur Trennung hinzu. `subdir` und `name` machen aus `new` eine vollständige Dateispezifikation.

```
  if (subdir[0] != '\\')                /* nicht ab Wurzelverz.?   */
  { new[2] = '\\';
    xgetcurdir (1 + new[0] - 'A', new + 3);
    /* nicht Wurzelverzeichnis, Name folgt?                        */
    if (new[3] && name[0])
    { strcat (new, "\\");
    };
  };
  strcat (new, subdir);                 /* vollst. Namen aufbauen  */
  strcat (new, name);
```

Nächster Schritt ist die Entfernung der relativen Verzeichnisangaben "`.`" und "`..`". Der einfache Punkt wird einfach überlesen; beim doppelten Punkt geht es ein Verzeichnis zurück, soweit das möglich ist. Schauen wir uns diesen Vorgang genauer an. `scan` durchläuft `new` von Verzeichnis zu Verzeichnis. Das gerade bearbeitete Verzeichnis wird in `subdir` und an die durch `to` bezeichnete Stelle in `new` kopiert. Man könnte zwar auch zwei verschiedene Strings als Quelle und Ziel verwenden, aber da `scan` stets größer oder gleich `to` ist, kommt es innerhalb von `new` zu keinem Konflikt.

```
  to   = new + 3;                       /* to, scan = Anfang Verz. */
  scan = to;
  do
  { /* naechstes Verzeichnis in SUBDIR und NEW kopieren           */
    sub = subdir;
    while ((*scan != '\\') && *scan)    /* bis Ende Teilpfad       */
    { *(sub++) = *scan;
      *(to++)  = *scan;
      scan++;
    };
    *sub = '\0';
```

[8] Anmerkung zum Code: Wenn `name` leer ist, ist wg. der Arbeitsweise von `split_fspec` auch `subdir` leer; d.h. die Überprüfung, ob evtl. `subdir` Zeichen enthält, kann dann entfallen.

Ist der erste Buchstabe von `subdir` ein Punkt, muß das gerade angehängte relative Verzeichnis wieder entfernt werden. Ist der zweite Buchstabe ebenfalls ein Punkt und in `new` noch ein Verzeichnis enthalten, geht es ein weiteres Verzeichnis zurück.

```
    /* relative Verzeichnisangaben entfernen                    */
    if (subdir[0] == '.')
    { while (*(--to) != '\\');          /* letztes Verz. entfernen */
      if (*(to - 1) != ':')             /* noch ein Verzeichnis da?*/
      { if (subdir[1] == '.')           /* zurueck (zu Vaterverz.)?*/
        { while (*(--to) != '\\');      /* letztes Verz. entfernen */
        };
      };
    };
```

Falls `scan` nicht auf das Ende (d.h. das abschließende 0-Byte) von `new` zeigt, folgt noch ein Verzeichnis oder Dateiname, der mit einem Backslash vom restlichen String zu trennen ist. Der Vorgang wird wiederholt, bis alle Zeichen bearbeitet sind.

```
    if (*scan)                          /* Backslash anhaengen     */
    { *(to++) = '\\';
      scan++;
    };
  } while (*scan);
  *to = '\0';
```

Falls `new` nur aus einer Laufwerksangabe besteht, wird noch der Backslash zur Kennzeichnung des Wurzelverzeichnisses ergänzt. Zur dieser Situation kann es kommen, wenn eine Pfadangabe nur relative Verzeichnisse enthält, die alle, inklusive dem Backslash des Wurzelverzeichnisses, entfernt werden.

```
  if (new[2] == '\0')
  { new[2] = '\\';
    new[3] = '\0';
  };
  strcpy (f_spec, new);
  return (0);
};
```

Funktion "Normalize Path 2":
`void norm_path2 (char f_spec[])`

Aufrufparameter:

`f_spec`	zu normalisierende Dateispezifikation/Pfad

Seiteneffekte:

`f_spec`	wird normalisiert

Rückgabewert:
keiner

Tabelle 4.12: `norm_path2`: Normalisiere Pfad 2

norm_path. Es gibt auch einen alternativen Weg, der die Hauptarbeit dem Kernel überläßt, aber wieder andere Nachteile mit sich bringt. Die Idee: Die Turbo-C Funk-

tion `getcurdir` bestimmt für ein angegebenes Laufwerk das aktuelle Verzeichnis ab Wurzelverzeichnis. Man versucht nun, den zu normalisierenden Pfad mit `chdir` zum aktuellen Pfad zu machen und fragt das Ergebnis der Operation ab.

Zuvor ist der aktuelle Pfad in `cur_dir` zu retten, der durch unseren Test verändert wird. Als erster Schritt wird das Laufwerk `drive` bestimmt, auf das sich die Dateispezifikation `f_spec` bezieht. Ohne explizite Angabe wird das aktuelle Laufwerk verwendet.

```
int norm_path (char f_spec[])
{ int result;                          /* Result der Teilanalyse  */
  char cur_dir[65];                    /* akt. Verzeichnis        */
  char test_dir[65], test_name[13];    /* Unterv./Name Teilanalyse*/
  char drive;                          /* Laufwerksnr. F_SPEC     */
  char pos;                            /* Index in String         */

  /* Laufwerksnummer von F_SPEC ermitteln                         */
  if (f_spec[1] == ':')
  { drive = f_spec[0] - ((f_spec[0] < 96) ? 'A' : 'a');
  }
  else
  { drive = xgetdisk ();
  };

  /* aktuelles Verzeichnis retten                                 */
  cur_dir[0] = 'A' + drive;            /* Laufwerksang. aufbauen  */
  cur_dir[1] = ':';
  cur_dir[2] = '\\';
  if (xgetcurdir (1 + drive, &cur_dir[3]))
  { return (NR_ERROR);
  };
```

Dann wird versucht, mit `chdir` auf `f_spec` zu wechseln. Für den Fall, daß `f_spec` leer ist oder nur eine Laufwerksangabe wie `A:` enthält, wird der reservierte Name für das aktuelle Verzeichnis "." hinzugefügt. Führt der Wechsel zum Erfolg, handelt es sich um eine Pfadangabe.

```
  /* F_SPEC leer oder nur Laufwerk?                               */
  if (f_spec[0] == '\0' || ((f_spec[1] == ':') && (f_spec[2] == '\0')))
  { strcat (f_spec, ".");
  };

  /* Test: ist F_SPEC Unterverzeichnis?                           */
  if (xchdir (f_spec) == 0)
  { result = NR_PATH;
  }
  else
```

Führt der Wechsel zu keinem Erfolg, muß geprüft werden, ob es sich bei `f_spec` um eine Dateispezifikation, d.h. eine Kombination von Pfad und Dateinamen, handelt. Dazu wird `f_spec` in die Teile `test_dir` und `test_name` zerlegt. Der abschließende Backslash von `test_dir` wird entfernt, falls es sich nicht um das Wurzelverzeichnis handelt. Schließlich wird analog zu dem schon beschriebenen Verfahren ein Versuch unternommen, auf `test_dir` zu wechseln. Schlägt auch dieser fehl, ist in `f_spec` ein illegaler Pfad enthalten (`result` = 3).

```
  { /* Test: existiert Unterverzeichnis in F_SPEC?               */
    split_fspec (SM_PATH, f_spec, test_dir);
```

```
    split_fspec (SM_NAME, f_spec, test_name);
    pos = strlen (test_dir) - 1;
    if ((test_dir[pos] == '\\') && (pos > 0) &&
        (test_dir[pos - 1] != ':'))
    { test_dir[pos] = '\0';
    };
    if ((test_dir[0] == '\0') ||
        (test_dir[1] == ':') && (test_dir[2] == '\0'))
    { strcat (test_dir, ".");
    };
    result = (xchdir (test_dir)) ? NR_INVAL : NR_FSPEC;
  };
```

Abschließend wird der aktuelle Pfad, den das Kernel ja in Normalform liefert, in `f_spec` übertragen. Ist `result` = `NR_FSPEC`, muß `f_spec` noch um den zuvor abgetrennten Dateinamen `test_name` ergänzt werden.

```
    /* normalisierten Pfad aufbauen                                  */
    f_spec[0] = 'A' + drive;
    f_spec[1] = ':';
    f_spec[2] = '\\';
    xgetcurdir (1 + drive, &f_spec[3]);
    if (result == NR_FSPEC)              /* F_SPEC war Dateispez.?  */
    { comp_fspec (f_spec, test_name);
    };
```

Anschließend ist noch der ursprüngliche aktuelle Pfad wieder herzustellen.

```
    /* aktuelles Verzeichnis restaurieren                            */
    if (xchdir (cur_dir))
    { result = NR_ERROR;                 /* sollte nicht passieren! */
    };
    return (result);
  };
```

Funktion "Normalize Path":
`int norm_path (char f_spec[])`

Aufrufparameter:

`f_spec`	zu normalisierende Dateispezifikation/Pfad

Seiteneffekte:

`f_spec`	wird normalisiert

Rückgabewert:
-1, `NR_ERROR`: Fehler; 1, `NR_PATH`: Pfad; 2, `NR_FSPEC`: Dateispezifikation; 3, `NR_INVAL`: unzulässiger Pfad

Tabelle 4.13: `norm_path`: Normalisiere Pfad

Ist keine Diskette eingelegt oder die Laufwerksklappe offen, führt die Bestimmung des aktuellen Verzeichnisses bei beiden Versionen zu einem Fehler. Das ist insofern nicht weiter tragisch, als daß der Zugriff auf die spezifizierte Datei aus dem gleichen Grund ebenfalls zu einem Fehler führen würde. `norm_path` führt deshalb nicht zu einem unnötigen Versagen des aufrufenden Programms.

4.4.3 Deaktivierung interner Kommandos

Interne Kommandos können über eine relativ aufwendige Filtershell oder durch externe Ersatz-Programme mit Kontrollfunktionen überwacht werden. Beim zweiten Verfahren sind die internen Befehle zunächst in externe umzuwandeln. Dabei hilft ein Blick in den internen Aufbau von `command.com` weiter. Dort findet sich nämlich eine Tabelle der internen Befehle, welche die Shell zur Bearbeitung von Eingaben des Benutzers durchsucht. Um z.B. den Befehl `copy` zu deaktivieren, muß der zugehörige Eintrag unkenntlich gemacht werden ("patchen"; engl. to patch = flicken). Dazu reicht bereits die Eingabe des gleichen Textes in Kleinbuchstaben aus. Diesen Vorgang wollen wir in Zukunft als "Externisieren" bezeichnen.

Im Folgenden ein kurzes Beispiel, das die Externisierung von `copy` mit Hilfe von `symdeb` zeigt.

```
symdeb command.com                    ; starten SYMDEB, laden Shell
s 100 100+cx "COPY"                   ; nach Text "COPY" suchen,
                                      ; Ergebnis der Suche sei <adr>
eb <adr> "copy"                       ; Eintrag ueberschreiben
w                                     ; Abspeichern
q                                     ; Quit
```

Das solchermaßen modifizierte `command.com` erkennt den Befehl `copy` nicht mehr und versucht, ein Programm gleichen Namens zu laden. Hier kommt `AVCopy` ins Spiel, das in `copy` umzubenennen ist. Analog ist mit den internen Befehlen `ren` *und* `rename` für `AVRename` zu verfahren. Wer möchte, kann weitere interne Befehle gegen externe Befehle mit Kontrollfunktionen austauschen. Damit die Transportkontrolle nicht unterlaufen werden kann, sind die externen Kommandos `xcopy`, `replace`, `restore` und ggf. weitere Kopier- und Umbenennungsprogramme aus dem System zu entfernen.

4.5 Realisierung des residenten Teils

4.5.1 Das Interrupt/"C"-Interface Std_INTC

Die speicherresidente Installierung eines Programms ist nicht weiter schwierig. Nach Aufruf der Funktion "Terminate and Stay Resident" wird das Programm beendet, aber nicht aus dem Speicher entfernt. In dieser Form haben wir lediglich ein Stück RAM kunstvoll verschwendet. Es fehlt der Anschluß an Interrupts, die ISRs des Programms aktivieren. In diesem recht umfangreichen Abschnitt werden wir uns mit dem Interrupt-"C"-Interface beschäftigen. Ziel ist die Verwendung normaler "C"-Funktionen als Serviceroutinen für Interruptaufrufe.

Problem: Beendigung der ISR. Der Versuch, eine konventionelle "C"-Funktion als ISR zu benutzen, führt beim Versuch der Rückkehr zum aufrufenden Programm zum Absturz. Das liegt daran, daß bei einem Interrupt drei Worte auf dem Stack abgelegt werden, nämlich `CS`-, `IP`- und Flagregister zum Zeitpunkt der Unterbrechung. Da unsere NEAR-Funktion (Modell `TINY`!) aber einen NEAR-Rücksprung verwendet, der

auf dem Stack nur das gerettete IP-Register erwartet, ist die Katastrophe im Sinne des Wortes vorprogrammiert.

Problem: Erhaltung der Registerinhalte. Das zweite Problem besteht darin, daß selbst bei erfolgreichem Aussprung wahrscheinlich alle Registerinhalte durch die Ausführung der ISR zerstört werden. Davon bemerkt das unterbrochene Programm nichts und arbeitet deshalb mit den falschen Werten weiter.

Das Retten und Restaurieren von Prozessorregistern ist auf Maschinenspracheebene recht einfach. Registerinhalte werden mit `push <Register>` auf dem Stack deponiert und mit `pop <Register>` wieder vom Stack geholt. Das Flagregister wird vom Prozessor automatisch auf den Stack gerettet. Die Deklaration einer Turbo-C-Funktion als `interrupt` bewirkt, daß der Compiler am Eingang und Ausgang der Funktion automatisch Codesequenzen zum Retten und Restaurieren der Register einfügt sowie das `iret`-Kommando verwendet. Das hat aber wieder den Nachteil, daß wir keine Werte an das aufrufende Programm zurückgeben können, weil die Registerinhalte vor dem Aussprung mit den Originalwerten überschrieben werden.

Problem: Benutzung des Stack. Davon abgesehen, daß nicht jeder "C"-Compiler über diese Fähigkeit verfügt, stellt die Verwaltung des Stack ein weiteres Problem dar. Falls die ISR

1. lokale Variablen verwendet oder
2. Unterprogramme in Anspruch nimmt, die u.U. weitere Unterprogramme aufrufen und evtl.
3. Parameter an diese Unterroutinen übergibt,

wird in jedem der drei Fälle Speicherplatz auf dem Stack verbraucht. Dieser ist durch die Unterbrechung (Rücksprungadresse, Flagregister) und das Retten der CPU-Register bereits vorbelastet. Da die ISR nicht wissen kann, wie groß der Stackbereich des unterbrochenen Programms ist, den sie ja implizit benutzt, empfiehlt es sich, auf einen eigenen Stack umzuschalten. Der Stackbereich kann z.B. durch eine globale oder als `static` deklarierte Variable des TSR-Programms reserviert werden. Zur Umschaltung auf den eigenen Stack sind der Inhalt des `SS`- und `SP`-Registers zu retten und auf das Ende des neuen Stack zu setzen. Scheinbar merkwürdig "Auf das Ende" deshalb, weil der Stack in Richtung der niedrigen Adressen wächst.

Problem: Reentrancy. Mit der Umschaltung des Stackbereichs ist folgendes Szenario vorstellbar. Ein überwachter Interrupt wird aufgerufen, durchläuft die Stackumschaltung und tritt in die Kontroll-ISR ein. Das Sicherheitsprogramm gibt z.B. etwas auf eine Logdatei aus und macht dabei selbst wieder von Funktionen des überwachten Interrupts Gebrauch. Erneut wird auf den gleichen Stack umgeschaltet und die Daten des ersten Durchlaufs überschrieben. Falls die ISR außerdem bei *jedem* Durchlauf den überwachten Interrupt aufruft, kommt es zu einer Endlosschleife.

Die angeführten Schwierigkeiten sind unter dem Begriff *Reentrancy*, "Wiedereintrittsfestigkeit", bekannt. "Reentrant" sind Programme, die zur gleichen Zeit mehrfach aufgerufen werden können. Besonders bei Multitasking-Betriebssystemen läßt sich auf

diese Weise Speicher sparen. Mehrere Programme können z.B. eine Unterroutine, die nur einmal im Speicher liegen muß, gemeinsam zur gleichen Zeit benutzen.

Wie wird unsere Kontroll-ISR reentrant? Abhilfe schafft das `in_isr`-Flag, das am Anfang der ISR abgefragt wird. Ist es gelöscht, darf die Kontrollroutine, der kritische Abschnitt, betreten werden und das Flag wird gesetzt. Ist das `in_isr`-Flag gesetzt, geht die Kontrolle an die alte ISR ohne Kontrolle über. Das ist statthaft, weil in diesem Fall nur das Sicherheitsprogramm den Interrupt ausgelöst haben kann (Annahme: Die Kontroll-ISR wird nicht durch andere, asynchrone[9] Interrupts unterbrochen). Beim Verlassen des kritischen Abschnitts wird `in_isr` wieder gelöscht.

Die Lösung: Std_INTC. Das Interrupt-"C"-Interface `Std_INTC` realisiert Mechanismen zur Vermeidung der oben genannten Probleme. Für `<nr>` ist im ganzen Quelltext die Nummer des betreffenden Interrupts einzusetzen. Ein Textverarbeitungsprogramm mit der Funktion "Search & Replace" ("suchen und ersetzen") kann hier nützlich sein.

Die Interface-Routine beginnt mit einer Reihe von `EXTERN`-Deklarationen, die dem Assembler anzeigen, daß die so bezeichneten Funktionen und Daten in anderen Modulen definiert werden. Dort sind die ISRs als gewöhnliche Funktionen realisiert (Name: `isr_<nr>`) und die Daten global deklariert, was einer Deklaration als `PUBLIC` entspricht. Die Schlüsselworte `BYTE`, `WORD` und `DWORD` entsprechen den Definitionen unserer Standardbibliothek. `NEAR` steht für einen NEAR-Zeiger oder eine Funktion, die über einen NEAR-Sprung erreicht werden soll. Der Assembler weiß dann, welche Sorte `ret`-Befehl er verwenden muß.

```
EXTRN _brkdis:NEAR                    ; Unterbrechung verhindern
EXTRN _brken:NEAR                     ; Unterbrechung zulassen
EXTRN _in_isr:BYTE                    ; "in ISR"-Flag
EXTRN _isr_<nr>:NEAR                  ; INT <nr> Service Routine
EXTRN _new_stack:NEAR                 ; Zeiger auf neuen Stack
EXTRN _old_int<nr>:DWORD              ; alter INT <nr> Vektor
EXTRN _old_ss:WORD                    ; altes Stacksegment
EXTRN _old_sp:WORD                    ; alter Stackpointer
```

Ein Ausschnitt aus `Std_TSR` (Besprechung im nächsten Abschnitt) zeigt, wie die Deklarationen auf der "C"-Seite aussehen.

```
/* globale und externe Variablen                              */
volatile BYTE in_isr;                 /* "in ISR"-Flag        */
BYTE    new_stack[512];               /* neuer Stack          */
void interrupt (far *old_int<nr>) ();  /* alter INT 0x<nr>-Vektor */
WORD    old_ss;                       /* alter Stackpointer   */
WORD    old_sp;
```

Die Deklaration des Codesegments ist uns bereits aus 3.2 "Grundlagen Hochsprachen" vertraut. Die `ASSUME`-Anweisung in der nächsten Zeile ist etwas schwierig zu erklären. Falls im Quelltext ein symbolischer Name wie `_in_isr` vorkommt, weiß der Assembler, in welchem Segment das Symbol liegt. Er weiß aber nicht, welches Register die Basis dieses Segments enthält, d.h. ob und welches "Segment Override Prefix" zu verwenden ist. Hier könnte der Programmierer aushelfen, in dem er selbst und evtl.

[9] Z.B. der System-Timer

fehlerträchtig für die korrekte Angabe des Segments sorgt. Alternativ dazu kann er über die ASSUME-Anweisung dem Assembler mitteilen, welches Segmentregister *er* mit welcher Basis belegt hat. Noch einmal: Der ASSUME-Befehl gibt dem Assembler lediglich einen *Hinweis* auf die Belegung der Segmentregister. Der Programmierer ist selbst dafür verantwortlich, daß die Angaben auch stimmen und muß die Register mit den korrekten Werten füllen.

```
_TEXT SEGMENT WORD PUBLIC 'CODE'
ASSUME CS:_TEXT, DS:_TEXT, ES:_TEXT, SS:_TEXT
```

Die als "NEAR" deklarierte Funktion _int<nr>, die Einsprungstelle der ISR, wird mit dem Schlüsselwort PUBLIC externen Modulen zugänglich gemacht. Die ISR beginnt mit der Sicherung der Inhalte aller Register auf den Stack. Die Nummern in den Kommentaren geben die Position relativ zum Wert des Registers SP an, den dieses nach Ablauf aller Sicherungsmaßnahmen besitzt. Den Grund dafür lernen wir später kennen. Die ersten drei Worte auf dem Stack, das Flag-, CS- und IP-Register, werden automatisch bei der Unterbrechung von der CPU dort hinterlegt.

```
PUBLIC _int<nr>

_int<nr> proc near
  ; Flags                              ; 24
  ; CS                                 ; 22
  ; IP                                 ; 20
  pushf                                ; 18 alle Register retten
  push  ax                             ; 16 (auch Statuswort, weil
  push  bx                             ; 14 bei Aussprung ueber
  push  cx                             ; 12 "cont" kein "iret"
  push  dx                             ; 10 ausgefuehrt wird)
  push  ds                             ;  8
  push  es                             ;  6
  push  bp                             ;  4
  push  si                             ;  2
  push  di                             ;  0
```

Anschließend wird das in_isr-Flag untersucht. Ist es gesetzt, befindet sich bereits ein Programm im kritischen Abschnitt und es wird zur Marke cont verzweigt. Dort werden alle Register wieder vom Stack restauriert und die Ausführung mit der Original-ISR fortgesetzt, deren Anfangsadresse in der Zeigervariable _old_int<nr> steht. Die Deklaration dword ptr zeigt dem Assembler an, daß der Inhalt von _old_int<nr> als FAR-Adresse zu interpretieren ist. Der iret-Befehl, der hier irgendwie zu fehlen scheint, steht am Ende der alten Serviceroutine und bewirkt die Rückkehr zum aufrufenden Programm.

```
  cmp cs:_in_isr, 00h                  ; "in_isr"-Flag geloescht?
  jne cont                             ; Nein: Kontroll-ISR umgehen

>> Code siehe folgender Text, "cont:" wurde vorgezogen <<

cont:                                  ; Operation validiert
  pop   di                             ; Register wiederherstellen
  pop   si
  pop   bp
  pop   es
  pop   ds
```

```
        pop   dx
        pop   cx
        pop   bx
        pop   ax
        popf
        jmp   dword ptr cs:[_old_int<nr>]     ; mit Original-ISR fortfahren

_int<nr> ENDP                                 ; Ende der Funktion
_TEXT ENDS                                    ; Ende des Code-Segments
      END                                     ; Ende des Quelltexts
```

Ist das `in_isr`-Flag gelöscht, wird es gesetzt und der momentane Stand des FAR-Zeigers `SS:SP` gerettet. Der anschließende Wechsel des Stackbereichs ist von den Befehlen `cli` (**cl**ear **i**nterrupt flag; Interrupt sperren) und `sti` (se**t** **i**nterrupt flag; Interrupt freigeben) umrahmt. Sie verhindern eine Unterbrechung zwischen dem Wechsel des Stacksegments und des Stackpointers durch z.B. den Timer-Interrupt. Die Folgen wären katastrophal, denn `SS:SP` würde in willkürliche Speicherbereiche zeigen. Die Ablage der Rücksprungadresse und des Flagregisters könnte wahre Verheerungen anrichten. Durch die Realisierung als echter kritischer Abschnitt wird der Stackwechsel abgesichert. Der Zusatz `OFFSET` vor `_new_stack` bewirkt, daß nicht der Inhalt des ersten Wortes, sondern die Startadresse von `_new_stack` eingesetzt wird. Auf die Bedeutung der Unterprogramme `brkdis` und `brkend` kommen wir noch zu sprechen.

```
        mov   cs:_in_isr, 0FFh                ; Ja: setze "in_isr"-Flag
        call  _brkdis                         ; Unterbrechung verhindern

        mov   cs:_old_ss, ss                  ; Stackzeiger retten
        mov   cs:_old_sp, sp
        mov   bp, cs                          ; auf neuen Stack umschalten
        cli
        mov   ss, bp
        mov   sp, OFFSET _new_stack + 512
        sti
```

Der "C"-Prototyp der vom Interrupt-Interface aufgerufenen ISR sieht folgendermaßen aus:

```
int isr_<nr> (void far *return_adr,
              WORD ax, WORD bx, WORD cx, WORD dx,
              WORD si, WORD di, WORD ds, WORD es);
```

Mit Hilfe der Daten-, Index- und Segmentregister kann die Kontrollroutine die Parameter des Aufrufs auswerten. Eine Spezialität ist der FAR-Zeiger `return_adr`, der die auf dem Stack hinterlegte Rücksprungadresse enthält. Diese zeigt in das Programm, das die Unterbrechung ausgelöst hat. Mit Hilfe der Funktionen `get_mcb_owner` und `get_name_mcb` läßt sich der Name des aufrufenden Programms bestimmen, was für Kontrollzwecke von großer Bedeutung ist.

```
        push  es                              ; Parameter auf Stack ablegen
        push  ds
        push  di
        push  si
        push  dx
        push  cx
        push  bx
        push  ax
```

```
mov   bp, cs:[_old_sp]                ; Ruecksprungadresse ablegen
mov   es, cs:[_old_ss]
push  es:[bp + 22]                    ; Offset siehe Kommentare
push  es:[bp + 20]                    ; neben Startsequenz
```

Der `call`-Befehl ruft die in "C" geschriebene Kontrollfunktion auf. Zuvor werden `DS` und `ES` mit `CS` gleichgesetzt, damit die Bedingungen des Speichermodells `TINY` erfüllt sind. Dazu noch mehr unter "Anmerkungen zur Funktion" weiter unten. Der Weg über den Stack ist erforderlich, weil es keine `mov`-Befehle für das `CS`-Register[10] oder den Transfer zwischen zwei Segmentregistern gibt.

```
push cs                               ; DS, ES = CS
pop ds                                ; (wg. Modell "TINY")
push cs
pop es
call _isr_<nr>                        ; ISR aufrufen
```

Nach der Rückkehr vom Funktionsaufruf müßte normalerweise der Wert 20 zum Stackpointer addiert werden, um den Platz für die Funktionsparameter (10 Worte sind $2 * 10 = 20$ Bytes) freizugeben. Durch die Umschaltung auf den alten Stack kann dieser Schritt entfallen.

```
cli                                   ; auf alten Stack schalten
mov   sp, cs:_old_sp
mov   ss, cs:_old_ss
sti

call  _brken                          ; Unterbrechung zulassen
mov   cs:_in_isr, 00h                 ; "in_isr"-Flag loeschen
```

Der Rückgabewert der Kontrollfunktion, der in `AX` steht, dient zwei Zwecken. Der Wert FFFF_{16} (äquivalent zu -1 in Wortbreite) signalisiert dem Interface, daß der Aufruf zulässig ist. Die Fortsetzung mit der Original-ISR erfolgt durch den Sprung zu `cont`.

```
cmp ax, 0FFFFh                        ; Funktionswert = 0xFFFF?
je  cont                              ; Ja: Operation validiert
```

Die Anforderung ist unzulässig, falls die Kontrollfunktion einen von FFFF_{16} verschiedenen Wert zurückgibt. Der Inhalt von `AX` entspricht dann der zu simulierenden Fehlernummer, die angeforderte Funktion wird nicht ausgeführt, die ISR also abgebrochen. Wie signalisiert die Kontroll-ISR dem aufrufenden Programm einen Fehler?

Kernelaufrufe geben im Fehlerfall im `AX`-Register die Fehlernummer zurück und setzen das Carry-Flag. Die Belegung des `AX`-Registers beim Aussprung aus der ISR bereitet uns kein Kopfzerbrechen, wohl aber das Setzen des Carry-Flags. Bei der Ausführung des `iret`-Kommandos werden nämlich das Registerpaar `CS:IP` sowie das Flagregister vom Stack zurückgeladen. Jegliche vorausgegangene Manipulation an irgendwelchen Flags bleibt damit wirkungslos. Der Trick besteht darin, das Flagregister *auf dem Stack* zu verändern. Beim Rücksprung wird das "gefälschte" Statuswort übernommen, und unser Ziel ist erreicht. Als Hilfe geben die Zahlen neben den `pop`-Befehlen den Offset relativ zum aktuellen Stand des Stackpointers `SP` an. Mit der Marke `stop` beginnt der Zweig, der den Aussprung mit simuliertem Fehler realisiert.

[10] Dadurch würde der Programmablauf verändert.

```
stop:                                   ; Nein: Abbruch ISR
  mov   bp, sp
  or    word ptr [bp + 24], 0001h       ; Carry-Flag in Flagregister
                                        ; auf Stack setzen
                                        ; (Fehler simulieren)
  pop   di                              ;  0 Register restaurieren
  pop   si                              ;  2
  pop   bp                              ;  4
  pop   es                              ;  6
  pop   ds                              ;  8
  pop   dx                              ; 10
  pop   cx                              ; 12
  pop   bx                              ; 14
  add   sp, 4h                          ; 16, 18: AX und PSW nicht
                                        ; 20: IP
                                        ; 22: CS
                                        ; 24: PSW
  iret                                  ; ISR beenden

>> Fortsetzung mit "cont:", siehe oben                          <<
```

Anmerkungen zur Funktion des Interfaces. Beim ersten Start des Installationsprogramms zeigen alle Segmentregister auf die Startadresse des PSP, wie es bei `com`-Programmen Sitte ist. Der Linker hat beim Bindevorgang alle Offsets (NEAR-Adressierung!) für Unterprogrammaufrufe und Datenzugriffe relativ zur dieser Basis berechnet. Wie sieht die Sache aber bei einem Aufruf der ISR durch einen Interrupt aus? Die Ausführung startet ja gar nicht mehr am Programmanfang, sondern beginnt bei der Serviceroutine. Kann das Interface überhaupt in dieser Form funktionieren?

Wenn wir bei der Installation den zu kontrollierenden Interruptvektor auf unsere Routine verstellen, behalten wir das aktuelle Codesegment bei. Das bedeutet, daß der `CS`-Teil des neuen Vektors auf die Basis unseres Programms zeigt und `IP` relativ dazu auf den Start der ISR. Genau so ist die Situation nach erfolgter Unterbrechung, denn die CPU setzt `CS:IP` auf den Start der Serviceroutine. Die Inhalte aller anderen Register sind unbestimmt. Aus diesem Grund sind das `DS`- und `ES`-Register auf den gleichen Wert wie `CS` zu setzen, um die Voraussetzungen des `TINY`-Modells zu erfüllen.

Erläuterungen. `brkdis` (Break Disable) verhindert, daß ein Programm durch Drücken der Tastenkombinationen `Ctrl` `C` oder `Ctrl` `Break` unterbrochen werden kann. Ganz wichtig wird das bei den Kontroll-ISRs, die unbedingt zu Ende geführt werden müssen. Geschieht das nicht, bleibt das `in_isr`-Flag gesetzt und kein Aufruf wird mehr kontrolliert. Auch die Reaktivierung oder Deinstallierung ist nicht möglich, es sei denn, man sucht und löscht das Flag mit einem Debugger im Speicher.

Das Kernel ruft bei einer Unterbrechung durch die oben genannten Tastendrücke den Interrupt 23_{16} auf, um dem Anwender die Möglichkeit zur Reaktion zu geben. `brkdis` setzt diesen Vektor auf die Routine `ctrl_stop` um, der alte Wert wird in `ctrl_off` und `ctrl_seg` zwischengespeichert. Das `DS`-Register ist unbedingt zu retten, weil "C"-Programme davon ausgehen, daß mindestens dieser Wert für die Datenadressierung erhalten bleibt. Gleiches gilt für die Register `SI` und `DI`, falls Turbo-C Registervariablen verwendet. Diese werden entweder explizit als `register` deklariert oder implizit benutzt, falls die Optimierung auf Geschwindigkeit eingeschaltet ist.

`brkdis` und `brken` manipulieren die Tabelle der Interruptvektoren, ohne Kernelfunktionen zu verwenden. Diese — etwas anrüchige — Methode wird verwendet, weil das Kernel beim Aufruf einer Funktion prüft, ob das laufende Programm abgebrochen werden soll. Falls jemand während der Abarbeitung der Kontrollfunktion eine entsprechende Taste gedrückt hat, würde die ISR verlassen, *bevor* das `in_isr`-Flag gelöscht werden konnte.

```
PUBLIC _brkdis
PUBLIC _brken

DGROUP GROUP _DATA

_DATA SEGMENT WORD PUBLIC 'DATA'
  ctrl_off      DW ?
  ctrl_seg      DW ?
_DATA ENDS

_TEXT SEGMENT BYTE PUBLIC 'CODE'
ASSUME cs:_TEXT, cs:DGROUP

_brkdis PROC NEAR
  push  ax
  push  ds
  push  es

  mov   ax, 0000h                       ; alten Vektor retten
  mov   es, ax                          ; ES = 0x0000
  mov   ax, es:[4*23h]                  ; 4 * 0x23 = Offset Eintrag
  mov   cs:ctrl_off, ax                 ; fuer ISR 0x23
  mov   ax, es:[4*23h+2]
  mov   cs:ctrl_seg, ax
  mov   ax, offset ctrl_stop            ; neuen Vektor setzen
  mov   es:[4*23h], ax
  push  cs
  pop   ds
  mov   es:[4*23h+2], ds

  pop   es
  pop   ds
  pop   ax
  ret
_brkdis ENDP
```

Der neue Break-Handler ist recht einfach gehalten. Es wird von unserer Seite nichts unternommen und dem Kernel signalisiert, daß die Break-Anforderung zu ignorieren ist.

```
ctrl_stop:
  iret
```

`brken` (Break Enable) ist das Gegenstück zu `brkdis` und restauriert den alten Vektor.

```
_brken PROC NEAR
  push  ax
  push  es

  mov   ax, 0000h                       ; Vektor restaurieren
  mov   es, ax
  mov   ax, cs:ctrl_off
```

```
  mov   es:[4*23h], ax
  mov   ax, cs:ctrl_seg
  mov   es:[4*23h+2], ax

  pop   es
  pop   ax
  ret
_brken ENDP
_TEXT ENDS
END
```

4.5.2 Die TSR-Plattform Std_TSR

Problemstellung. Basis aller Watcher des Programmpakets AV-System bildet eine residente Programmplattform als Träger für die Kontrollfunktionen. Diese sollte als speicherresident installierbares "TSR"-Programm konzipiert werden, um den Programmieraufwand gering zu halten. Aus dem gleichen Grund soll die Implementation so weitgehend wie möglich in der Sprache "C" erfolgen. Für die Kontrolle von Interruptaufrufen sind eigene Serviceroutinen notwendig, deren Programmierung jedoch nicht von jedem "C"-Compiler und in der von uns benötigten Form unterstützt wird. Das Interrupt/"C"-Interface `Std_INTC` ermöglicht uns, trotzdem wesentliche Teile der ISRs in "C" zu realisieren.

Aufgabenbeschreibung. Ziel der Entwicklung ist die TSR-Plattform `Std_TSR` (Standard-TSR), die sich für die Realisierung beliebiger speicherresidenter Programme eignet und die folgende Grundfunktionen realisiert:

- Eine evtl. bereits bestehende *Installation erkennen* und dementsprechend das Programm abbrechen.
- Programm resident *installieren*, dabei Initialisierungsroutine ausführen (Interruptvektoren umsetzen, Speicher freigeben etc.).
- Kommunikation per Interruptaufruf mit anderen Programmen, um Funktionen zu aktivieren und Daten auszutauschen.
- Programm auf Anforderung (gegen Unbefugte gesichert) *deinstallieren*, dabei Restaurierung durchführen (Interruptvektoren auf Originalwerte zurücksetzen etc.).

Eine wichtige Vorgabe ist die Größe des Programms. Das TSR-Programm sollte möglichst klein ausfallen, damit ein Maximum an Arbeitsspeicher für andere, residente und transiente Programme zur Verfügung steht. Aus diesem Grund werden TSR-Programme meist ausschließlich in Assembler geschrieben, weil dies den kompaktesten Code ergibt.

An dieser Stelle sei eine generelle Anmerkung zu den residenten Programmen in diesem Buch erlaubt. Die Forderung nach möglichst kleinen TSR-Programmen legt es nahe, diese komplett in Maschinensprache zu implementieren. Die Programmierung in Assembler hat aber viele Nachteile, darunter

- die Komplexität des Codes,

- die große Anzahl von Befehlen, die notwendig ist, um selbst simple Funktionen zu realisieren,
- die Unübersichtlichkeit und
- die schlechte Wartbarkeit des Ergebnisses.

Daher wurde zugunsten der besseren Verständlichkeit und Mitverfolgbarkeit auf die maschinennahe Hochsprache "C" zurückgegriffen. Ziel des Buches ist es, Algorithmen und Verfahren zu entwerfen und in einfachen, beispielhaften Programmen zu implementieren. Die entworfenen, funktionsfähigen Algorithmen lassen sich natürlich auch in andere Sprachen übertragen. Die Erstellung professioneller, kommerziell tauglicher Software kann auf dieser Basis erfolgen, würde aber den hier gesetzten Rahmen sprengen.

Systemarchitektur. Aus der Aufgabenbeschreibung lassen sich bereits unmittelbar einzelne Funktionen ableiten. Die Startfunktion `main` muß eine evtl. bereits bestehende Installation erkennen und das Programm in diesem Fall beenden. Sonst würde mehrfach Speicher reserviert und unerwünschte Nebeneffekte könnten die Funktionsfähigkeit des Watchers beeinträchtigen. Zur residenten Installierung gehört die Freigabe von nicht benötigten Programmteilen (Environment), die Reservierung von Datenbereichen (Puffer etc.) und schließlich die Feststellung des Speicherbedarfs. Vor der Installierung, unter MS-DOS identisch mit dem Verlassen des Programms, sind noch von `init` Variablen zu initialisieren und die zu überwachenden Interruptvektoren auf eigene ISRs umzusetzen.

Der Watcher ist danach nur noch über Softwareinterrupts ansprechbar. Das bedeutet, daß alle Funktionen zur Deinstallation, Datenaustausch usw. Bestandteil einer Service-ISR sein müssen, in die sich der Watcher eingeklinkt hat. Zwei Arten des Aufrufs sind möglich: Entweder über einen zuvor unbenutzten Interrupt oder über unbelegte Funktionsnummern eines schon anderweitig verwendeten. `Std_TSR` macht von der zweiten Methode Gebrauch und fügt dem DOS-Interrupt 21_{16} neuen Funktionen hinzu, die dem internen Service dienen. Das vorgeschlagene Verfahren schlägt zwei Fliegen mit einer Klappe, weil sich der Watcher sowieso zu Überwachungszwecken in die Kernel-Funktionen einschalten muß. Auf diese Weise wird Platz gespart — eine einzige ISR dient den zwei Aufgaben Kontrolle und Programmservice.

Die über die "C"-Funktion `intercom` aufrufbaren Servicefunktionen dienen verschiedenen Zwecken. In Registern können Werte, z.B. Zeiger auf Datenbereiche, übergeben und damit Nachrichten ausgetauscht werden. So kann `main` bei Start des Programms die Anwesenheit einer installierten Kopie feststellen. Bei der Deinstallation sind alle übernommenen Interruptvektoren durch `restore` auf ihre ursprünglichen Werte zurückzusetzen. Danach kann der durch das Programm belegte Speicher freigegeben werden. Die Servicefunktionen sind gegen unbefugte Aktivierung zu schützen. Die Absicherung erfolgt durch ein im Quelltext festgelegtes Paßwort (4 Bytes), das bei jedem Aufruf überprüft wird.

Problemstellung (intercom). Die durch `Std_INTC` praktizierte Parameterübergabe über Register hat den Nachteil, daß kein echter Datenaustausch mit anderen

Programmen möglich ist. Das reicht zwar aus, um zwischen DOS- und Servicefunktionen sowie Subfunktionen zu unterscheiden, aber nicht, um z.B. ein längeres Paßwort zu übergeben. Die momentane Situation erlaubt es jedem, der die Funktionsnummer der Servicedienste kennt, diese zu aktivieren. Es muß aber dafür gesorgt sein, daß z.B. Benutzer eines Rechenzentrums die Schutzprogramme nicht einfach ausschalten können.

Weiterhin ist eine Situation denkbar, bei der mehrere TSR-Programme des von uns entwickelten Typs gleichzeitig aktiv sind. Bei Serviceanforderungen ist dann auf geeignete Weise zwischen den einzelnen Programmen zu unterscheiden. Da der Aufruf über einen Softwareinterrupt erfolgt, ist das Ziel unbestimmt. Wenn mehrere residente Programme denselben Interrupt übernommen haben, wandert der Aufruf von einer ISR zur nächsten. Jedes Programm der Kette muß die Zieladresse prüfen und den Aufruf entweder weitergeben oder bei Übereinstimmung selbst bearbeiten.

Aufgabenbeschreibung (intercom). Aus den angeführten Gründen ist eine Funktion `intercom` vorzusehen, welche die Servicefunktion der TSR-Plattform mit bestimmten Parametern aufruft und einen Statuswert zurückmeldet. Am flexibelsten ist die Übergabe eines Zeigers, weil dadurch die Struktur der Daten beliebig ist (Zeiger auf `int`, auf Arrays, auf `structs` etc.). Um durch Modifikation des Zeigers Daten zurückgeben zu können, übergeben wir einen Zeiger auf einen Zeiger. Die Bezeichnung "Zeiger auf Zeiger" ist weniger kompliziert als sie sich vielleicht anhört, wie die Grafik 4.9 verdeutlicht. `ptr` ist ein FAR-Zeiger auf einen weiteren FAR-Zeiger `data`, der auf die zu übergebenden Daten zeigt. Mit `ptr` übergeben wir der ISR die Adresse von `data`. Falls Daten an das aufrufende Programm zurückgegeben werden sollen, kann die ISR über `ptr` die Adresse der Daten in `data` eintragen (Tab. 4.14).

Abbildung 4.9: Zeiger auf Zeiger auf Datenobjekt

Funktionsbeschreibung (intercom). Wie der folgende Quellcode zeigt, ist das Verfahren simpel. Der FAR-Zeiger `ptr` wird in `DS:DX` übertragen. Analog zu normalen Kernelfunktionen enthält `AH` die Funktionsnummer ($C4_{16}$, willkürlicher Wert) und `AL` die Nummer der Subfunktion `subfunc`. Rückgabewert ist der Funktionswert der Servicefunktion, der sich bereits in `AX` befindet.

```
param STRUC
  old_bp        DW ?
  ip            DW ?
  subfunc       DW ?
  ptr_seg       DW ?
  ptr_off       DW ?
param ENDS

PUBLIC _intercom
```

Funktion "Intercom" `intercom`**:**
`WORD intercom (BYTE subfunc, void far * far *ptr)`
Aufrufparameter:

`ptr`	FAR-Zeiger auf FAR-Zeiger auf Daten
`subfunc`	Subfunktionsnummer

Seiteneffekte:
keine

Rückgabewert:
Rückgabewert der Servicefunktion

Tabelle 4.14: `intercom`: Kommunikation mit residentem Programm

```
_TEXT SEGMENT PUBLIC BYTE 'CODE'
ASSUME CS:_TEXT
  _intercom PROC NEAR
  push bp                                ; Register retten
  mov bp, sp
  push ds
  push dx
  mov ds, [bp].ptr_off                   ; Register laden, Aufruf
  mov dx, [bp].ptr_seg
  mov al, byte ptr [bp].subfunc
  mov ah, 0C4h
  int 21h
  pop dx                                 ; Register restaurieren
  pop ds
  pop bp
  ret
  _intercom  ENDP
_TEXT ENDS
END
```

Anwendung (intercom). Zunächst ist ein Nachrichtenformat zu definieren, mit dem externe und residente Programme miteinander kommunizieren. Die Nachricht umfaßt immer die Identifikationsnummer des residenten Programms (Zieladresse), ein Paßwort, die Nummer der Servicefunktion und evtl. weitere Daten. Die Servicefunktion ist durch den Parameter `subfunc` direkt bestimmt; der Datenblock, auf den `ptr` indirekt verweist, enthält die anderen Informationen:

```
struct T_I_MESSAGE
{ BYTE  prg_id;                          ; Zieladresse
  DWORD pwd;                             ; Passwort
};
```

Das führt uns zum nächsten Punkt: Welche Servicefunktionen sind erforderlich (symbolische Konstanten in Klammern)? Benötigt werden

- Paßwort ungültig (`IM_INVAL_PWD`). Paßwort in Nachricht und internes Paßwort stimmen nicht überein.

- Abfrage auf Installation (`IM_RUTHERE` von "are you there?"). Vermeidung von Doppelinstallationen.
- Anforderung Deinstallation (`IM_DEINST`). Für Wartungszwecke, die durch das TSR-Programm behindert würden.
- Funktionsnummer unbekannt (`IM_UNKNOWN`). Die angeforderte Servicefunktion wird nicht unterstützt.

Später werden noch weitere Funktionen hinzukommen, die sich mit der Übergabe von Daten befassen.

Bei einem `intercom`-Aufruf prüft die ISR zunächst, ob die Nachricht an die eigene Adresse gerichtet ist. Falls nicht, wird der Aufruf ähnlich dem Token-Ring-Verfahren über den ursprünglichen Interruptvektor weitergereicht. Damit erhält das nächste Programm der Interruptkette die Chance, die Nachricht anzunehmen. `message` gibt eine Nachricht auf dem Bildschirm aus, ohne den Inhalt zu zerstören.

```
/* einfuegen in "Defines"; Bedeutung s. Text                        */
#define IM_RUTHERE      0x00
#define IM_DEINST       0x01
#define IM_I_TABLE      0x02
#define IM_INVAL_PWD    0x04
#define IM_UNKNOWN      0xFF
#define PRG_ID          ('t' << 8)      /* Identifikationsnummer   */
#define PRG_NAME        "STD_TSR"
#define PWD             0x123456781     /* Password                */

/* globale und externe Variablen                                    */
extern WORD __brklvl;                   /* fuer Groessenbestimmung */
volatile BYTE in_isr;                   /* "in ISR"-Flag           */
BYTE    new_stack[512];                 /* neuer Stack             */
void interrupt (far *old_int21) ();     /* alter INT 0x21-Vektor   */
WORD    old_psp;                        /* alte Segmentadresse PSP */
WORD    old_ss;                         /* alter Stackpointer      */
WORD    old_sp;

WORD isr_21 (void far *ret_adr,
             WORD ax, WORD bx, WORD cx, WORD dx,
             WORD si, WORD di, WORD ds, WORD es)
{ struct T_I_MESSAGE far *msg_ptr;      /* Zeiger auf Nachricht    */

  /* interner Service angefordert?                                  */
  if ((ax & 0xFF00) == 0xC400)
  { msg_ptr = *(struct T_I_MESSAGE far * far *)MAKE_FARPTR (ds, dx);
    /* Nachricht fuer dieses Programm?                              */
    if (msg_ptr -> prg_id != (PRG_ID >> 8))
    { message (0x5F, "Not for me");
      return (0xFFFF);
    };
```

Die einfachste Dienstleistung, `IM_RUTHERE`, besteht lediglich aus einer Ausführungsmeldung, die sich aus der Programmnummer `PRG_ID` im höheren Byte und der Funktions- oder Meldungsnummer im unteren Byte zusammensetzt. Dieses Schema gilt allgemein für Rückmeldungen der Servicefunktion.

```
    if ((ax & 0x00FF) == IM_RUTHERE)    /* Anfrage:"Are you there?"*/
```

```
    { return (PRG_ID | IM_RUTHERE);     /* Ja!                      */
    };
```

Nächster Schritt ist die Überprüfung des Paßworts, einer 4-Byte-Zahl (ca. 4 Milliarden Kombinationen). Stimmt das Paßwort mit dem internen Paßwort des residenten Programms überein, wird die Servicefunktion ausgeführt. Ist das Paßwort unzulässig, wird eine entsprechende Meldung zurückgegeben.

```
    if (msg_ptr -> pwd != PWD)          /* Passwort korrekt?        */
    { message (0x5F, "Wrong password");
      return (PRG_ID | IM_INVAL_PWD);
    };
```

Die Servicefunktion besteht aus einer `switch`-Anweisung, die von der Subfunktionsnummer abhängig ist. Bei der Deinstallierung werden zunächst mit `restore` alle Interruptvektoren auf ihre ursprünglichen Werte zurückgesetzt. Danach kann der durch das Programm belegte Speicher freigegeben werden.

```
    /* bearbeite Subfunktionen                                      */
    switch (ax & 0x00FF)
    {
>> Hier nach Bedarf Code fuer weitere Funktionen einfuegen         <<
      default:
        return (PRG_ID | IM_UNKNOWN);
    };
  };
  /* Hier eigene Routinen einfuegen                                */
  return (0xFFFF);                      /* mit Org.-ISR fortfahren */
};
```

Detektion. Was ist, wenn das Programm bereits speicherresident installiert ist und nochmals aufgerufen wird? Zur Detektion der Installation gibt es mehrere Möglichkeiten [17]. Entweder stellt man die Anwesenheit des Programms im Speicher fest, indem man die Kette der MCBs durchforstet oder implementiert wie in unserem Fall im residenten Teil eine Funktion, die auf Abfrage von außen reagiert.

```
int main (void)
{ struct T_I_MESSAGE msg;               /* Nachricht                */
  struct T_I_MESSAGE far *msg_ptr;      /* Zeiger auf Nachricht     */

  /* Schon installiert?                                            */
  msg.prg_id = PRG_ID >> 8;
  msg_ptr = (struct T_I_MESSAGE far *)&msg;
  if (intercom (IM_RUTHERE, (void far *)&msg_ptr) ==
      (PRG_ID | IM_RUTHERE))
  { message (0x1F, "Allready installed");
    return  (1);
  };
```

Freigabe Environment. Wenn das TSR-Programm das Environment nicht benötigt, kann es dies mit der Funktion "Release Memory Block" (in "C" `freemem`; Tab. A.12) freigeben. Das funktioniert, weil das Environment in einem ganz normalen Speicherblock untergebracht ist. Die als Parameter benötigte Segmentadresse des Environments enthalten wir über das Wort bei Offset $2C_{16}$ des PSP. Die Segmentadresse des PSP wiederum liefert die Funktion "Get PSP Address" (`getpsp`; Tab. A.18). Dem Kernel ist es übrigens egal, ob ein Programm einen Speicherblock freigibt, der ihm gar nicht

gehört — unter "richtigen" Betriebssystemen undenkbar. So könnte ein Programm die Liste der MCBs durchgehen und jeden Block mit fatalen Folgen freigeben.

```
/* Environment freigeben                                          */
if (xfreemem (*(WORD far *)MAKE_FARPTR (xgetpsp (), 0x2C)))
{ message (0x9F, "Couldn't free environment");
  return  (2);
};
```

Installierung. Zur Installierung benötigt die Funktion `keep` die Programmgröße in Paragraphs. Zum Speicherbedarf eines Programms zählen der eigentliche Programmcode, Daten und der Stackbereich, falls wir später Funktionsaufrufe durchführen und lokale Variablen benutzen wollen. Weil die "C"-Bibliotheken keine Funktion in der Art "`get_program_size`" oder dergleichen zur Verfügung stellen, sind alternative Methoden zur Platzbestimmung erforderlich. Gesucht wird ein Hinweis auf die letzte durch das Programm belegte Speicheradresse. In einem Assemblerprogramm genügt es, hinter Code-, Daten- und Stackbereichen ein Label `last_byte` zu vereinbaren. Bei Hochsprachen geben die Bibliotheken und der Compiler die Reihenfolge der Module unsichtbar vor. Der Anwender hat keinen Einfluß auf die spätere Lage seiner Module oder evtl. vereinbarter Bezeichner. Ein Blick in den Aufbau der Speichermodelle und die Struktur von kompilierten "C"-Programmen gibt wichtige Hinweise, doch müssen wir dafür etwas weiter ausholen.

Zunächst ist ein bestimmtes Speichermodell festzulegen, weil davon alle weiteren Untersuchungen abhängig sind. Wir wählen das Modell `TINY`, um einen möglichst einfachen Programmaufbau zu erhalten und weil der Watcher sicher weniger als 64 kB Speicher umfassen wird. Das Ergebnis der Kompilierung ist ein `exe`-Programm, das vor Benutzung noch mit dem externen Kommando `exe2bin` in eine `com`-Datei umzuwandeln ist. Den Grund dafür deutet die Warnung des Linkers "`no stack`" an. Im Modell `TINY` ist kein separater Stackbereich vorgesehen, wie ihn alle `exe`-Programme haben und beim Start automatisch benutzen. `exe2bin` entfernt den überflüssigen `exe`-Header, der u.a. dem Lader von MS-DOS Aufschluß über die Lage des Stack gibt, und transformiert die Datei in einen verwendbaren Zustand.

Beim Start eines `com`-Programms reserviert MS-DOS zunächst allen verfügbaren Speicher, jedoch höchstens 64kB, weil dies die maximale Segmentgröße ist. Alle Segmentregister zeigen auf den Start des Speicherblocks, in dem sich Programm, Daten und Stack in einem einzigen Segment befinden. Der Stack wächst vom Ende des Speicherblocks her in Richtung Anfang und macht durch seine Lage eine Verkleinerung zunächst unmöglich, da sonst Teile des Stack freigegeben würden. Der standardmäßig festgelegte Stackbereich muß so verlegt werden, daß auch bei maximaler Stackbenutzung kein Programmcode oder Daten überschrieben werden, was fatal wäre.

Diese und andere Aufgaben übernimmt der Startup-Code des Compilers, der vor dem Anwenderprogramm durchlaufen wird. Im Startup-Modul werden auch die Reihenfolge der Segmente sowie einige globale Namen, Variablen und Funktionen definiert. Einige Werte stehen schon bei der Kompilierung oder beim Einladen in den Speicher fest, andere werden erst zur Laufzeit berechnet. Abb. 4.10 zeigt, welche Bezeichner zu welchem Zeitpunkt welche Adressen und Speicherbereiche referenzieren.

Abbildung 4.10: Aufbau Startcode für das Speichermodell `TINY` (Turbo-C)

Unter Turbo-C beginnt der sog. "Heap" (engl.: Stapel, Haufen), von dem mittels `malloc` (**m**emory **alloc**ate) Speicher reserviert werden kann, unmittelbar nach Ende des Programms. Die Größe ist durch die Konstante `__heaplen` festgelegt, die wiederum in der Turbo-Bibliothek oder im Anwenderprogramm definiert ist[11]. Während der Heap in Richtung der hohen Adressen wächst, kommt ihm der Stack "von oben" entgegen. Der Wert von `__stklen` bestimmt analog zu `__heaplen` die Größe des Stack.

Weil unsere ISR einen eigenen Stackbereich `new_stack` anstatt des Programmstack benutzt, kann dieser bei der Installierung komplett freigegeben werden. Es wäre nun nützlich, etwas über die Endadresse des Heaps zu erfahren, aber auch für diese Abfrage gibt es in "C" keine eigene Funktion. Zwei Wege führen ab jetzt zum gewünschten Resultat. Der eine ist eher allgemein gehalten und geht davon aus, daß jeder "C"-Compiler Programme mit dem Aufbau Code - Daten - Heap - Stack generiert. Der andere basiert speziell auf Turbo-C und liefert das kürzeste Verfahren.

`malloc` belegt das *erste* freie Stück Speicher, das die gewünschte Größe hat, oder meldet einen Fehler zurück. Dem aufrufenden Programm wird im Erfolgsfall die Startadresse des neu reservierten Blocks übergeben, und das ist der springende Punkt. Ein Aufruf von `malloc` mit einer Blockgröße von eins liefert die momentane Endadresse des Heaps zurück. Da diese relativ zu `DS` und damit zum PSP angegeben wird, wissen

[11] Die Angabe des Anwenders hat Vorrang, weil sie im Linkprozeß später auftaucht.

wir nun die Länge des Programms. Die Blockgröße von einem Byte ist notwendig, weil `malloc` sonst einen Fehler meldet.

Die andere Methode kommt ohne Aufruf der Funktion `malloc` aus, die eine ganze Reihe Speicherplatz fressender Funktionen nach sich zieht. `___brklvl` (drei Unterstriche) zeigt nämlich höchst komfortabel auf das Ende des Heaps. Die Deklaration `extern WORD __brklvl` (zwei Unterstriche) und eine einfache Zuweisungen erledigen die Bedarfsbestimmung im Handumdrehen. Der Verlust eines Unterstrichs kommt durch die Eigenschaft des Compilers zustande, jedem Symbol im Quelltext einen Unterstrich voranzustellen.

Doch Vorsicht: Alle Speicherreservierungen müssen *vor* der Längenberechnung getätigt werden, weil sonst die Länge des Datenbereichs und damit des Programms *nachträglich* anwachsen würde. Beim Aufruf der Funktion "Terminate and Stay Resident" würde Speicher freigegeben, von dem das Programm annimmt, er sei reserviert.

init führt alle notwendigen Initialisierungen durch. Dazu gehört vor allen Dingen das Umsetzen von Interruptvektoren auf eigene ISRs. In `old_psp` wird die Segmentadresse des PSP hinterlegt, um bei der Deinstallierung den durch das Programm belegten Speicher freigeben zu können. Die ermittelte Länge des Programms in Bytes muß noch durch 16 geteilt werden, um die Anzahl der zu reservierenden Paragraphs zu erhalten. Bei der Division ohne Rest durch 16 fallen bis zu 15 Bytes unter den Tisch. Dies wird durch die Erhöhung um einen Paragraph sicher ausgeglichen.

```
  /* Initialisieren                                                  */
  if (init ())
  { message (0x9F, "Initialisation error");
    return  (3);
  };

  /* resident installieren, Ende des Programms!                      */
  xkeep (0, 1 + (__brklvl >> 4));
  /* Warnung "Function should return value" ignorieren               */
};

int init (void)
{ in_isr = 0;                          /* Start mit "pruefen ein" */
  /* Hier eigene Initialisierungsroutinen einfuegen                  */
  old_psp = xgetpsp ();
  /* alten INT 0x21-Vektor retten / neue ISR installieren            */
  old_int21 = xgetvect (0x21);
  xsetvect (0x21, (void interrupt (far *) ())
    MAKE_FARPTR (old_psp, (unsigned)int21));
  return (0);
};
```

Deinstallierung. Bei der Deinstallierung ist darauf zu achten, daß vor der Freigabe von Environment (evtl. schon bei Installation) und Programm alle Interruptvektoren wieder ihre ursprünglichen Werte erhalten [9]. Geschieht das nicht, werden weiterhin die ISRs im mittlerweile für andere Programme verfügbaren Speicher benutzt. Das nächste Programm, das geladen wird, könnte den Bereich des ehemaligen TSR-Programms und damit auch die ISRs mit fatalen Folgen überschreiben. Die Aufforderung zur Deinstallierung wird von `intercom` an eine neue Funktion der Serviceroutine übermittelt:

```
/* einfuegen in "Defines"                                           */
#define PRG_NAME        "STD_TSR"

/* in Service-Funktion, Abteilung Subfunktionen, einfuegen          */
```

```
case IM_DEINST:                    /* deinstallieren          */
  restore ();
  message (0x5F, PRG_NAME " deinstalled...");
  if (xfreemem (old_psp))
  { message (0xDF, "Couldn't free memory");
  };
  return (PRG_ID | IM_DEINST);
```

restore macht alle durch `init` durchgeführten Operationen rückgängig. Das bedeutet insbesondere, daß alle Interruptvektoren ihre ursprünglichen Werte erhalten.

```
int restore (void)
{ /* Hier eigene Restaurierungsfunktionen einfuegen              */
  /* alten INT 0x21-Vektor wiederherstellen                      */
  xsetvect (0x21, old_int21);
  return (0);
};
```

4.5.3 TSR-Programme und Datenzugriff

Residente Programme können auf vier Arten an Informationen wie Referenzdaten, Dateirechte, Partitionsrechte und globale Rechte gelangen. Die folgende Aufstellung ist nach Grad der Auslagerung der Daten und der Lademechanismen von "so intern wie möglich" bis "so extern wie möglich" gegliedert:

1. Integration der Daten in den Quelltext,
2. Laden von Datei beim Start mit internem Lader, interne Speicherung,
3. Laden von Datei mit externem Lader, interne Speicherung,
4. Laden von Datei bei Bedarf mit internem Lader.

Das erste Verfahren eignet sich nur für unveränderliche Daten und scheidet aus, wenn der Anwender das Programm nicht selbst für die eigenen Anforderungen modifizieren und kompilieren kann. Für einige globale Rechte, wie das Ändern von Datum und Uhrzeit (vergleichsweise selten benutzte Funktionen), könnten Standardwerte, sinnvollerweise Verbote, angenommen werden.

Option zwei erfordert, daß die Daten während der Laufzeit des Programms unveränderlich sind, denn die Informationen werden nur einmal beim Start des Programms gelesen. Danach könnte die Laderoutine samt dem Code für die Initialisierungsroutine abgeworfen werden, was bei "C"-Programmen aber nicht möglich ist (unbekannte Anordnung der Segmente und Bezeichner).

Nummer drei ist eine Modifikation von Punkt zwei, die das Problem mit dem Abwurf der Laderoutine unter "C" elegant löst. Nur einmal benötigte Programmteile werden in einem externen Programm, z.B. `AVConfig`, untergebracht. Mit einer `intercom`-Anfrage an den Watcher ermittelt `AVConfig` die Adresse der internen Tabelle. Die notwendigen Daten werden durch beliebig komfortable und aufwendige Routinen ermittelt und einfach übertragen. Falls die Daten während des Betriebs des Kontrollprogramms geändert werden sollen, kann ein Update auf die gleiche Weise wie bei der

ersten Initialisierung erfolgen. Das residente Programm kann auch selbst das externe Ladeprogramm quasi als Overlay anstoßen. Falls die Änderungen sehr häufig auftreten, wird die Ausführungszeit des Konfigurationsprogramms zu einem kritischen Faktor.

Bei großen Tabellen gerät jeder der ersten drei Vorschläge in Schwierigkeiten, weil die Daten im kostbaren Hauptspeicher abgelegt werden und den Raum für andere Programme beschränken. Methode Nummer vier ist die einzig praktikable Lösung bei großen und veränderlichen Datenmengen, scheidet aber bei zeitkritischen Anwendungen aus.

Fallbeispiel "Referenzliste"
Eine große Anzahl von Tabelleneinträgen liegt im Fall der Integritätsprüfung vor, bei der für jede zu überprüfende Datei ein Eintrag mit Name und diversen Attributen erforderlich ist (s. `ChkState`). 20 kB (Wert vom System des Autors) sind mehr, als der Watcher wahrscheinlich selbst verbraucht und mehr, als die meisten unter MS-DOS opfern möchten. Ein Zugriff auf die Referenzdaten findet nur beim Start eines Programms oder Lesezugriff auf eine sicherheitskritische Datei statt, d.h. nur relativ selten im Abstand von vielleicht Minuten. Eine Verzögerung von auch mehreren Sekunden fällt verglichen mit der Ladezeit des Programms nicht so sehr ins Gewicht. Wegen der großen Datenmenge und dem unkritischen Zeitverhalten wäre Lösung 4 zu empfehlen.

Fallbeispiel "Liste der Partitionsrechte"
Die Liste der Partitionsrechte wird zur Beurteilung der Zulässigkeit von BIOS-Disk-Aufrufen benötigt. Weil das Kernel, das wahrscheinlich die Diskfunktion aufgerufen hat, nicht reentrant ist, scheiden Dateizugriffe bei der Kontrolle von BIOS-Funktionen generell aus. Die Geschwindigkeit des Zugriffs ist ein weiterer Grund für die interne Speicherung. Jede Dateioperation über das Kernel löst eine Vielzahl von Diskzugriffen aus, die der Watcher jeder einzeln überprüfen muß. Die Benutzung externer Rechtelisten würde, selbst wenn sie möglich wäre, den Betrieb unerträglich verlangsamen und das Laufwerk durch permanente Spurwechsel ruinieren.

Tabelle 4.15 faßt die Ergebnisse unserer Untersuchung noch einmal knapp zusammen und ordnet sinnvolle Einsatzmöglichkeiten zu.

Verfahren	Datentyp	Zugriff	Umfang	Eignung
Permanente interne Liste	unveränderlich	schnell	gering	globale Rechte
Externe Datei, interner Lader und Liste	ab Start unveränderlich	schnell	gering	Partitions- und globale Rechte
Externe Datei und Lader, interne Liste	veränderlich	schnell	gering	Partitions- und globale Rechte
Externe Datei = Liste, interner Lader	veränderlich	langsam	groß	Referenzdaten, Dateirechte

Tabelle 4.15: Datenzugriff von TSR-Programmen

4.5.4 AVWatch

Problemstellung. Unter einem ungesicherten Betriebssystem wie MS-DOS können Softwareanomalien nach belieben tätig werden. Alle Typen verbrauchen durch ihre bloße Anwesenheit Ressourcen wie Speicherplatz und Rechenzeit und können darüber hinaus das System manipulieren. Ein Virus verursacht zusätzliche Schäden durch die Infektion von Programmen. Ermöglicht werden diese Aktivitäten dadurch, daß Aufrufe von Betriebssystemfunktionen keinerlei Kontrolle unterliegen.

Obwohl jede Softwareanomalie an einem bestimmten Punkt ins System gelangt sein muß, kann mangels Logdaten der Verursacher i.d.R. nicht mehr festgestellt werden. Ebensowenig lassen sich Verbreitungswege verfolgen, um z.B. die Ausdehnung des Schadens zu ermitteln, Teilsysteme abzuschotten oder Empfänger von verseuchter Software zu warnen.

Aufgabenbeschreibung. Zur Erfüllung der Anforderungen ist ein residentes Programm vorzusehen, das sich in relevante Interrupts einklinkt, die Aufrufparameter anhand einer Rechtedatei überprüft und entsprechend dem Ergebnis die Operation zuläßt oder abbricht. Die zu überwachenden Funktionen lassen sich in vier Gruppen nach dem Wirkungsbereich zusammenfassen:

- Global: Veränderung von Systemdatum und -zeit,
- Partition: Schreib-/Leseschutz für jegliche Daten oder nur Urladeinformation,
- Datei (passiv): Schreiben, Lesen, Ausführen, residente Installierung,
- Datei (aktiv): Überprüfung der Integrität.

Um Einbruchsversuche zu dokumentieren und evtl. den Täter zu identifizieren, werden spezifizierbare Operationen in Form einer Logdatei aufgezeichnet. Die Logdatei sowie alle Rechtedateien sind vor fremdem Zugriff zu schützen.

Systemarchitektur. Es werden vier separate Versionen von `AVWatch` erstellt, die jeweils eine der genannten Aufgabengruppen abdecken. Dadurch wird die Entwicklung einfacher und durchschaubarer; die Einzelprogramme sind leichter zu testen und können auch separat eingesetzt werden. Ebenso ist es möglich, die Funktionen verschiedener Watcher in einem Programm zu integrieren. Die Programme und ihre Funktionen sind (s.a. Abb. 4.11):

1. `AVWatchG`: Überwachung fixer globaler Rechte (keine Rechtedatei).
2. `AVWatchI`: Schutz der Programmintegrität mit Rückgriff auf die Referenzliste von `ChkState` (Implementation von Cohen's Betriebssystem S3).
3. `AVWatchP`: Überwachung von Sektorzugriffen (Partitionorientiert; Rechtedatei `p_rights.lst`).
4. `AVWatchF`: Überwachung von Dateizugriffen (Rechtedatei `f_rights.lst`).

Von den Quelltexten der Watcher werden nur die Teile abgedruckt, die zum Code von `Std_TSR` hinzukommen. Eine Kommentarzeile vor jedem Fragment gibt an, an welcher Stelle der Code einzufügen ist. Diese Punkte sind im Quelltext von `Std_TSR` markiert, soweit es die Erweiterung von Funktionen betrifft.

Abbildung 4.11: Ein-/Ausgabe `AVWatch*`

4.5.5 AVWatchG(lobal)

Die Überwachung einiger einfacher Funktionen dient uns als Vorübung für größere Aufgaben. Konkret geht es um die Kernel-Funktionen $2B_{16}$ "Set Date" und $2D_{16}$ "Set Time", deren Objekt stets die Systemuhr ist. Ein Rückgabewert von 0 signalisiert das erfolgreiche Setzen von Datum oder Zeit. Das aufrufende Programm, das Subjekt der Operation, bleibt unberücksichtigt; daher die Bezeichnung "globale" Rechte.

Funktion. Bei einem Kernelaufruf enthält das Register `AH` die Funktionsnummer. Ist `AH` von $2B_{16}$ und $2D_{16}$ verschieden, wird die Kontroll-ISR mit $FFFF_{16}$ beendet. Das Interrupt-"C"-Interface fährt dann mit der Ausführung der Original-ISR fort. Andernfalls bewirkt der Wert 0 den Abbruch des Aufrufs. Obwohl dem aufrufenden Programm der erfolgreiche Abschluß der Operation vorgetäuscht wird, bleibt die Systemuhr unverändert.

```
/* einfuegen in "Defines"                                          */
#define PRG_NAME        "AVWatch (G)"
#define PRG_ID          ('g' << 8)

/* einfuegen in ISR_21                                             */
  BYTE func;                            /* Funktionsnummer         */

  func = ax >> 8;
  /* SET DATE oder SET TIME?                                       */
```

```
if ((func == 0x2B) || (func == 0x2D))
{
  return (0x0000);                    /* simuliere "in Ordnung"  */
};
```

Nach Kompilierung, Konvertierung in eine com-Datei und Start ist von Anwesenheit und Wirkung nichts zu merken. Lediglich der freie Arbeitsspeicher ist um ein paar Kilobytes geschrumpft. Ein Test mit `date` oder `time` zeigt aber, daß `AVWatchG` funktioniert. Egal, ob die eingegebenen Werte gültig sind oder nicht, das Kommando vermeldet stets den ordnungsgemäßen Vollzug. Ein nochmaliger Aufruf zeigt, daß Zeit und Datum nicht verändert wurden — ein erstes, wenn auch kleines Erfolgserlebnis.

4.5.6 AVWatchI(ntegrity)

`AVWatchI` implementiert das von Cohen vorgeschlagene "Betriebssystem" S3, das auf der Überprüfung der Integrität von Programmen basiert. Dazu fängt die Kontroll-ISR die Kernel-Funktion $4B_{16}$ "Load and Execute" ab. Das Objekt, das auszuführende Programm, ist der Gegenstand der Kontrolle. Das Subjekt der Operation, das aufrufende Programm, bleibt immer noch unberücksichtigt.

Für die Berechnung der Signatur wird ein Puffer benötigt, der hier aus Platz- und Geschwindigkeitsgründen die Größe eines Sektors hat. `I_DEFAULT` und `I_INTERACT` bestimmen das Verhalten des Watchers, wie wir bei der Besprechung der ISR noch sehen werden. `init` initialisiert die CRC-Funktionen und vervollständigt die Namen der beiden von `ChkState` erzeugten Referenzdateien.

```
/* einfuegen in "Defines"                                          */
#define BUFSIZE         512             /* Puffergroesse           */
#define CRC_START       0x0000
#define I_DEFAULT       NO
#define I_INTERACT      YES
#define POLYNOMIAL      0xA001
#define PRG_NAME        "AVWatch (I)"
#define PRG_ID          ('i' << 8)

/* einfuegen in "globale und externe Variablen"                    */
BYTE    buf[BUFSIZE];                   /* Puffer f. CRC-Berechnung*/
char    f_subdir[65], f_file[65];       /* Namen d. Referenzdateien*/

/* einfuegen in INIT                                               */
  /* erzeuge vollst. Dateinamen, initialisiere CRC-Funktionen      */
  add_path (argv[0], "chkstate.sui", f_subdir);
  add_path (argv[0], "chkstate.fii", f_file);
  make_crc_table (POLYNOMIAL);
```

Kernstück von `AVWatchI` ist die ISR zur Kontrolle der Kernel-Aufrufe. Im Falle der Funktion $4B_{16}$ "Load and Execute" wird zunächst der durch das Registerpaar `DS:DX` referenzierte Programmname in den String `object` umkopiert. Dies ist notwendig, weil wir mit dem Speichermodell `TINY` arbeiten und deshalb alle Funktionen die Übergabe von NEAR-Zeigern erwarten. Die Übergabe eines FAR-Zeigers an `get_ref` würde im günstigsten Fall Fehlfunktionen, wahrscheinlich aber den Absturz des Programms hervorrufen.

```
/* einfuegen in ISR_21                                              */
  static char msg[80];                  /* Meldungspuffer          */
  static char object[65];               /* Programmname (Objekt)   */
  struct T_FILE file;                   /* Referenzeintrag Datei   */
  WORD sig;                             /* aktuelle Signatur       */
  int  i;                               /* Index in String         */
  int  f_nr;                            /* Nummer des Dateieintrags*/
  char far *text;                       /* f. NEAR/FAR-Umsetzung   */
  char decide, decide_old;              /* Entscheidungs-Flag      */
  char err;                             /* Status GET_REF          */

  if ((ax >> 8) == 0x4B)                /* "Load & Execute"?       */
  { text = MAKE_FARPTR (ds, dx);
    i = 0;                              /* FAR-String nach NEAR    */
    while ((object[i] = *(text++)) != '\0')
    { i++;
    };
```

`err` hält fest, ob ein passender Eintrag für das Programm in der Referenzdatei gefunden und gelesen werden konnte. Im Erfolgsfall wird die aktuelle Signatur `sig` für die Programmdatei ermittelt. Falls die Berechnung nicht erfolgen konnte, wird der Benutzer entsprechend informiert. In beiden Fehlerfällen behält `decide` seinen Wert von -1 bei. Ging bis hierher alles glatt, wird das Ergebnis des Vergleichs zwischen alter und aktueller Signatur `decide` zugewiesen. Falls die Werte nicht übereinstimmen, wird eine entsprechende Nachricht in `msg` geschrieben.

```
  decide = -1;
  /* lese Referenzeintrag fuer Programmdatei                       */
  if ((err = get_ref (object, &file, &f_nr, f_subdir, f_file))
       == 0)
  { /* berechne aktuelle Signatur                                  */
    sig = CRC_START;
    if (sig_crc (buf, BUFSIZE, object, &sig) == 0)
    { /* vergleiche mit Referenz-Signatur                          */
      if ((decide = (sig == file.sign[0])) == 0)
      { strcpy (msg, "Integrity violated. [U]pdate & Execute,");
      };
    }
    else
    { message (0xCF, "Couldn't compute signature");
      err = -6;
    };
  };
```

Ist `decide` kleiner 0, konnte entweder kein Eintrag gefunden oder die Signatur nicht berechnet werden. In diesem Fall tritt die Voreinstellung `I_DEFAULT` für `decide` in Kraft; `decide_old` nimmt dabei den alten Wert von `decide` auf. Gleichzeitig wird in `msg` vermerkt, daß aufgrund eines Fehlers keine Aussage über die Integrität des Programms gemacht werden kann. Von einer Fehlermeldung in Klartext wurde aus Platzgründen abgesehen. Ist `decide` gleich 1, ist der Aufruf zulässig und die Original-ISR wird ausgeführt. Ein Wert von 0 bedeutet, daß der Aufruf unzulässig ist.

```
  if ((decide_old = decide) < 0)
  { decide = I_DEFAULT;
    strcpy (msg, "Uncertain [0].");
    msg[11] -= err;                 /* Fehlernummer -> in MSG  */
  };
```

```
if (decide == 1)
{ return (0xFFFF);
};
```

Ist `I_INTERACT` gleich 0, wird die ISR sofort mit einem simuliertem "Datei nicht gefunden"-Fehler abgebrochen. Andernfalls wird der Benutzer über den Stand der Dinge informiert und nach seiner Entscheidung gefragt. Er kann die Ausführung abbrechen oder das Programm trotzdem starten und evtl. den Eintrag in der Referenzdatei auf den neuesten Stand bringen lassen. Letztere Option steht nur zur Verfügung, wenn kein Fehler beim Signaturvergleich aufgetreten ist, also ein Eintrag in der Referenzdatei existiert und die aktuelle Signatur berechnet werden konnte.

```
if (I_INTERACT)
{ strcat (msg, " [A]bort, [E]xecute ?");
  switch (message (0x4F, msg) & 0xFF)
  { case 'u':
      if (!decide_old)              /* Eintrag gefunden        */
      { file.sign[0] = sig;         /* Signatur aktualisieren  */
        put_ref (&file, f_nr, f_file);
      };
    case 'e':
      return (0xFFFF);
  };
};
return (0x0002);
};
```

Der Anwender sollte, bevor er trotz Warnmeldung ein Programm startet, auf jeden Fall wissen, warum die Programmdatei verändert wurde oder nicht in der Referenzliste gespeichert ist. Im Zweifelsfall empfiehlt es sich, die Ausführung abzubrechen und das Programm entweder zu untersuchen oder von einer Sicherheitskopie zu laden. Die von Cohen vorgesehene Option "Sicherheitskopie [automatisch] von Backup laden" wurde aus naheliegenden Gründen nicht implementiert und muß manuell durch den Anwender erfolgen. Für den Betrieb in "feindlicher" Umgebung (Rechenzentrum, Betrieb etc.) sollte `I_INTERACT` gleich 0 sein, damit kein Anwender modifizierte und evtl. verseuchte Programme starten kann.

4.5.7 AVWatchP(artition)

Mit der Überwachung von Sektorzugriffen steigen wir zwei Schichten in der Betriebssystemhierarchie auf die Ebene des BIOS hinab. Zwar bietet das DOS-Kernel neben den Datei- und Verzeichnisfunktionen auch Funktionen zum Lesen und Schreiben von absoluten Sektoren über die Interrupts 25_{16} und 26_{16} an. Da aber *alle* Zugriffe auf Diskette und Festplatte letztendlich über den BIOS-Interrupt 13_{16} abgewickelt werden, müssen wir nur diese Funktion mit einer zweiten Kontroll-ISR überwachen.

Problem: Die Partitionierung. Die Kernel-Funktionen zum Lesen und Schreiben von Sektoren benötigen als Parameter die logische Laufwerksnummer, den Startsektor, die Anzahl der Sektoren und die Adresse des Übergabepuffers. Der Hinweis auf die *logische* Laufwerksnummer zeigt, daß das Kernel Partitions der Festplatte sehr wohl

unterscheidet. Der Disk-Interrupt hingegen erwartet die Übergabe einer physikalischen Laufwerksnummer, also die Angabe eines konkreten Gerätes.

Beispiel "Zugriff auf Partitions"
Die erste Festplatte sei in die Partitions `C:` und `D:` unterteilt. Ein Programm greife über den Interrupt 25_{16} "Absolute Disk Read" auf beide Partitions lesend zu. Ein Kontrollprogramm, das sich in den Interrupt 13_{16} eingeklinkt hat, empfängt nur Aufrufe für Festplattenlaufwerk 0.

Auswertung der Partitionsdaten. Das geschilderte Verhalten wird für uns zu einem Problem, wenn wir z.B. softwaremäßig einen Schreibschutz für einzelne Partitions und nicht für ganze Festplatten vortäuschen wollen. Ein Programm, das alle Schreibzugriffe auf Festplatte 0 abblockt, verhindert schreibenden Zugriff auf alle Partitions dieses Laufwerks. Die Frage ist, wie man aus den Aufrufparametern der BIOS-Funktion Rückschlüsse auf die Partition ziehen kann. Die Lösung bringt die Auswertung des MBRs, der uns bereits in Abschnitt 3.5.3 begegnet ist. Dieser enthält nämlich Informationen darüber, welche Sektoren zu welcher Partition gehören.

Beim Start von `AVWatchP` legen wir aus diesem Grund eine Tabelle mit den Partitionsdaten an, auf welche die ISR für den Disk-Interrupt bei Aufrufen zurückgreifen kann. Theoretisch könnte man sich den Platz für die Partitionstabelle sparen und bei jedem Aufruf die Daten neu von Festplatte lesen, doch das kostet Zeit, die wir auf diesem Systemlevel nicht haben. Die Tabelle enthält für jede eingebaute Festplatte ein Verzeichnis, in das für jedes logische Laufwerk Start- und Endspur eingetragen sind (dazu später mehr). Da `AVWatchP` beim Start nicht bekannt ist, wieviele Festplatten mit wievielen Partitions installiert sind, muß die Tabelle flexibel, aber möglichst zeit- und platzsparend angelegt sein. Wie das Auslesen der Aufteilungsdaten funktioniert, wurde in Abschnitt 3.5.3 mit dem Programm `ReadPart` gezeigt. Eine genaue Beschreibung der Lesefunktion für `AVWatchP` findet sich in 4.5.10 "AVConfig" (Funktion `im_i_table`).

Die Wahl fällt auf ein fixes Array, das für maximal fünf Laufwerke mit insg. 10 Partitions ausgelegt ist. Die Bestimmung der Partition aus der Spurnummer ist eine Kombination von direktem Zugriff und sequentieller Suche. Die modifizierte physikalische Laufwerksnummer dient als Index in `start_part` (s. Funktionsbeschreibung). Der so ermittelte Wert liefert den Index des ersten Partitionseintrags in `part`, ab dem sequentiell gesucht werden muß. Der Speicherverbrauch von nur 65 Bytes rechtfertigt nicht den Aufwand einer dynamischen Verwaltung der Partitionsdaten. Damit hat die Partitionstabelle folgende Struktur:

```
struct T_I_TABLE
{ BYTE  start_part[5];                  /* max. 5 phys. Laufwerke  */
  struct T_P_ENTRY part[10];            /* max. 10 log. Laufwerke  */
};

struct T_P_ENTRY
{ WORD  start;                          /* erste Spur              */
  WORD  end;                            /* letzte Spur + 1         */
  WORD  rights;                         /* Partitionsrechte        */
};
```

In `start` und `end` ist eine *lineare Spuradresse* eingetragen, die sich aus Zylindernummer $*$ 16 (max. Anzahl Köpfe) + Kopfnummer ergibt. Damit wird vermieden, daß wir

zwei Angaben, nämlich Zylinder- und Kopfnummer, getrennt behandeln müssen. Die Nichtbeachtung der Sektornummer ist statthaft, weil `fdisk` die Festplatte so aufteilt, daß Partitionen stets an Spurgrenzen beginnen bzw. enden. Wäre das nicht der Fall, müßte in die Berechnung der Start-/Endadresse auch die Sektornummer einfließen.

RAM-Disks, CD-ROM- und Bandlaufwerke besitzen eigene Gerätetreiber, die sich nicht der BIOS-Funktionen bedienen. Darum lassen sich Zugriffe auf diese Geräte nicht durch Kontrolle des Interrupts 13_{16} abfangen. Die einzige Eingriffsmöglichkeit besteht in der Überwachung von Dateifunktionen auf Kernel-Ebene, der sich `AVWatchF` annehmen wird.

Zurück zur Kontroll-ISR. Zunächst sind einige allgemeine Umbauten an `Std_TSR` erforderlich, welche in erster Linie die neue ISR und die Tabelle der Partitionsdaten betreffen.

```
/* einfuegen in "Defines"                                          */
#define P_INTERACT      NO
#define PRG_NAME        "AVWatch (P)"
#define PRG_ID          ('p' << 8)

/* einfuegen in "Prototypen"                                       */
extern void int13       (void);         /* Interrupt-C-Interface   */
WORD    isr_13          (void far *ret_adr,
                         WORD ax, WORD bx, WORD cx, WORD dx,
                         WORD si, WORD di, WORD ds, WORD es);

/* einfuegen in "globale und externe Variablen"                    */
void interrupt (far *old_int13) ();     /* alter INT 0x13-Vektor   */
struct T_I_TABLE i_table =              /* Partitions-Information  */
{ 0, 0, 0, 0, 0,                        /* Startindizes            */
  {{0, 0xFFFF, 0xFFFF},                 /* Partitions-Tabelle      */
   {0, 0, 0xFFFF},
   {0, 0, 0xFFFF},
   {0, 0, 0xFFFF},
   {0, 0, 0xFFFF},
   {0, 0, 0xFFFF},
   {0, 0, 0xFFFF},
   {0, 0, 0xFFFF},
   {0, 0, 0xFFFF}}
};

/* einfuegen in INIT                                               */
  /* rette alten INT 0x13-Vektor / installiere neue ISR            */
  old_int13 = xgetvect (0x13);
  xsetvect (0x13, (void interrupt (far *) ())
            MAKE_FARPTR (old_psp, (unsigned)int13));

/* einfuegen in RESTORE                                            */
  /* restauriere alten INT 0x13-Vektor                             */
  xsetvect (0x13, old_int13);
```

Klassifizierung der Zugriffe. Jede Funktion des Disk-Interrupts wird zunächst klassifiziert (Belegung der Register s.Tab. A.24). Das Array `req_list` enthält Informationen darüber, welche Funktion (Nummer dient als Index) welche Rechte erfordert. In jedem Byte sind die Rechte bitweise verschlüsselt, wie Tabelle 4.16 zeigt. Sind keine Rechte erforderlich (Wert 0) oder untersteht die Funktion keiner Kontrolle (nicht im

Array enthalten), sorgt der Rückgabewert $FFFF_{16}$ für die normale Ausführung.

Bit	erforderliches Recht
0	Zugriff außerhalb Partition (z.B. Master Boot Record)
1	Zugriff auf Partition Boot Record
2	formatieren
3	schreiben
4	lesen

Tabelle 4.16: Bitmuster erforderliche Rechte unter `AVWatchP`

```
WORD isr_13 (void far *ret_adr,
             WORD ax, WORD bx, WORD cx, WORD dx,
             WORD si, WORD di, WORD ds, WORD es)
{ static char msg[81];                  /* Meldungspuffer          */
  WORD req_list[] =                     /* erforderliche Rechte    */
  { 0x0000, 0x0000, 0x0010, 0x0008, 0x0000, 0x000C, 0x000C, 0x000C,
    0x0000, 0x0000, 0x0010, 0x0008
  };
  WORD req;                             /* angeforderte Rechte     */
  WORD l_track;                         /* lineare Spuradresse     */
  BYTE func;                            /* Funktionsnummer         */
  BYTE p_drive;                         /* phys. Laufwerksnummer   */
  BYTE l_drive;                         /* log. Laufwerksnummer    */
  char decide;                          /* Entscheidungs-Flag      */

  /* nicht ueberwachte Funktion?                                   */
  func = ax >> 8;
  if (((req = req_list[func]) == 0) || (func > 0x0B))
  { return (0xFFFF);
  };
```

Unterliegt die Funktion der Kontrolle, wird zunächst die physikalische Laufwerksnummer `d_drive` festgestellt. Diese ist noch in die korrekte Eintragsnummer für die Partitionstabelle umzuwandeln. Die Nummern der Diskettenlaufwerke können direkt übernommen werden. Zugriffe auf Festplattenlaufwerke sind daran zu erkennen, daß Bit 7 der Laufwerksnummer in `DX` gesetzt ist. Durch Ausmaskieren von Bit 7 und Addition von zwei (= Anzahl der Einträge für Floppylaufwerke) erhält man Nummer des Festplatteneintrags.

```
  /* berechne phys. Laufwerksnummer                                */
  p_drive = dx & 0x007F;
  if (dx & 0x0080)                      /* Festplatte?             */
  { p_drive += 2;                       /* wg. 2 Floppy-Eintraegen */
  };
```

`d_drive` dient als Index in das Array `start_part`, das die Nummer des ersten logischen Laufwerks `l_drive` liefert. `l_drive` wird so lange erhöht, bis die aus Zylinder- und Kopf-Nummer ermittelte lineare Spuradresse `l_track` zwischen Start- und Endspur eines Tabelleneintrags liegt.

```
  /* berechne log. Laufwerksnummer                                 */
  l_drive = i_table.start_part[p_drive];
  l_track = (GET_TRACK (cx) << 4) + (dx >> 8);
  decide  = TRUE;
```

```
while (((l_track <  i_table.part[l_drive].start) ||
        (l_track >= i_table.part[l_drive].end)) && decide)
{ l_drive++;
  decide = (l_drive < i_table.start_part[p_drive + 1]);
};
```

Für den Fall, daß kein passender Eintrag gefunden werden konnte (`flag` gesetzt), gibt es zwei Erklärungen:

1. Falls sich die gesuchte Spur vor der ersten Spur der ersten Partition befindet, kann es sich nur um Daten zur Speicherverwaltung wie den MBR handeln, auf die ein normales Programm kein Zugriff haben sollte. Von der Bedeutung her ähnliches gilt für die PBRs der einzelnen Partitions, die u.U. ein Bootprogramm enthalten (s.u.).
2. Zugriffe auf Spuren zwischen Partitionen sind, falls das überhaupt von der Aufteilung der Festplatte her möglich ist, ebenfalls suspekt. Hier ist das Objekt der Operation undefiniert.

Da die PBRs im Gegensatz zum MBR Bestandteil der Partitions sind, ist eine separate Überprüfung notwendig, ob ein Zugriff auf diese Daten erfolgt. Der Test ist einfach zu realisieren, weil der Bootblock immer der erste logische Sektor eines Datenträgers oder einer Partition ist. Wegen der Bedrohung durch Bootviren ist der Schutz dieser Bereiche wünschenswert.

Da nun Operation und Subjekt feststehen, kann die Zulässigkeit der Operation überprüft werden.

```
if (!decide)                          /* ausserhalb Partition?   */
{ req |= 0x0001;                      /* L_DRIVE ungueltig!      */
  l_drive = i_table.start_part[p_drive];
}
else
{
  /* 1. Sektor = Partition Boot Record?                          */
  if ((l_track == i_table.part[l_drive].start) &&
      (GET_SECTOR (cx) == 1))
  { req |= 0x0002;
  };
};

/* pruefe Rechte                                                 */
if ((req & i_table.part[l_drive].rights) == req)
{ return (0xFFFF);
};
strcpy (msg, "Illegal direct sector access");
if (req & 0x0001)
{ strcat (msg, " (out of partition)");
};
if (req & 0x0002)
{ strcat (msg, " (boot record)");
};
```

Falls die Operation nicht zulässig ist und der Benutzer einschreiten darf, wird eine Meldung ausgegeben und der Anwender kann über das weitere Verfahren entscheiden.

```
  if (P_INTERACT)
  { strcat (msg, ". Proceed y/n?");
    if (message (0x4F, msg) == 'y')
    { return (0xFFFF);
    };
  };

  /* simuliere "Write Protect Error"                              */
  return (0x0300);
};
```

Aufbau der Rechtedatei. Wie werden Schutz von Partitionen, MBR und PBR festgelegt? Wir können zur Laufzeit auf keine Rechtedatei zugreifen, weil das viel zu langsam wäre und Reentrancy-Probleme nach sich zieht. Die Partitionsdaten ändern sich nach der Installation des Watchers nicht und brauchen nur einmal beim Start ausgewertet zu werden. Aufgrund unserer Untersuchung "TSR-Programme und Datenzugriff" wählen wir die Methode "Interne Liste, externer Lader".

Für jedes logische Laufwerk kann Lese-, Schreib- und Formatierschutz (**r**ead, **w**rite, **f**ormat) definiert werden. Dazu kommen noch der Zugriffsschutz für den PBR und MBR (**B**oot Record, **M**aster Boot Record). Zur Option "Schutz MBR" sind zwei Dinge anzumerken:

1. Dieser Schutz erstreckt sich auf *alle* Bereiche eines physikalischen Laufwerks, die keiner Partition angehören.
2. Die Option 'm' muß bei den Rechten der ersten Partition eines Laufwerks angegeben werden.

Zur Festlegung der Rechte benutzen wir wieder eine Textdatei, die durch eine Erweiterung von `AVConfig` in eine komprimierte Form umgesetzt und zu `AVWatchP` übertragen wird. Für die interne Speicherung der Verbote dient der Eintrag `BYTE rights` in `struct T_P_ENTRY`. Ein Eintrag in `p_rights.lst` ist folgendermaßen aufgebaut:

```
Eintrag   ::= <Laufwerk> <Rechte>
Laufwerk  ::= <Buchstabe>:
Buchstabe ::= A|B..Y|Z
Rechte    ::= {Option(en)}
Option    ::= r|w|f|b|m
```

Laden der Partitionstabelle. `AVConfig` fragt bei `AVWatchP` mit der Nachricht `IM_I_TABLE` die Adresse der Partitionstabelle `i_table` ab. Nach Auswertung der Datenträgerstrukturen und Umsetzung der Rechteliste schreibt das Konfigurationsprogramm die Ergebnisse direkt in die Tabelle von `AVWatchP` (s.a. 4.5.10 "`AVConfig`"). `AVWatchP` und `AVConfig` (Aufrufparameter "`p`") müssen nacheinander als Einheit ausgeführt werden, weil das System sonst ungeschützt bleibt.

```
/* in Service-Funktion von ISR_21, Abteilung Subfunktionen,
 * einfuegen
 */
      case IM_I_TABLE:
        *(void far * far *)MAKE_FARPTR (ds, dx) =
          (void far *)&i_table;
        return (PRG_ID | IM_I_TABLE);
```

Hinweis. Falls auf Ihrem Rechner das Lesen der Partitionsdaten aus irgendwelchen Gründen nicht korrekt funktioniert, können Sie die notwendigen Informationen mit dem externen Kommando `fdisk` bestimmen. Die Werte sind dann manuell in den Quellcode einzutragen, und zwar in die Initialisierung von `i_table`. In diesem Fall läßt sich Speicherplatz einsparen, weil die Größe der internen Tabelle den Verhältnissen genau angepaßt werden kann. Die Funktion `IM_I_TABLE` der `isr_21` ist dann überflüssig und wird ersatzlos gestrichen.

4.5.8 AVWatchF(ile)

Die Überwachung der Dateizugriffe ist der komplexeste Teil der Watcher-Serie. Dank umfangreicher Vorarbeiten in den vergangenen Kapiteln und Abschnitten ist der Rückgriff auf Funktionen möglich, die die Arbeit stark erleichtern und das Hauptprogramm übersichtlich halten.

Systemarchitektur. Die Aufgaben von `AVWatchF` gliedern sich in vier Abschnitte:

- Bestimmung des Subjekts (Benutzer bzw. stellvertretend das aufrufende Programm),
- Bestimmung der erforderlichen Rechte (abhängig von Operation),
- Bestimmung des Objekts (zu bearbeitende Datei),
- Bewertung der Zulässigkeit.

Das Rechtekonzept. Wir werden uns zuerst der Konzeption der Rechte zuwenden, weil dies einige Überlegungen erfordert und einen größeren Teil dieses Abschnitts in Anspruch nimmt. Auch dieser Problemkreis läßt sich in mehrere Unterpunkte gliedern:

- Art der Rechte (Lesen, Schreiben etc.),
- Interpretation der Rechte (Erlaubnis, Verweigerung, Verhalten bei Mehrdeutigkeiten) und
- Speicherung der Rechte (dynamische Verwaltung, Sicherheit).

Beispiel "UNIX"
Das von UNIX verwendete System ist recht flexibel und einfach zu handhaben, hat aber für unsere Zwecke einige Nachteile, die in der Implementierung und Anwendung bestehen (s.S. 46 "Dateirechte unter UNIX").

Problem: Verwaltung der Rechte. Die Dateirechte werden unter UNIX vom Betriebssystem verwaltet und sind fester Bestandteil des Dateisystems. Über Betriebssystemaufrufe können diese Rechte verändert werden, falls der Anwender dazu befugt ist. Systemprogramme laufen deshalb oft im Superuser-Modus ab und sorgen z.B. beim Kopieren und Umbenennen von Dateien für die korrekte Zuordnung von Rechten zu (neuen) Dateien. Unter MS-DOS müßte eine Liste aller Dateien und der zugehörigen

Rechte angelegt und verwaltet werden. Bei jedem Aufruf von `copy`, `rename` und anderen Kommandos wäre der Eintrag zu verschieben, zu ändern oder zu löschen — eine undankbare Aufgabe, die dem Aufwand gleichkommt, ein neues Betriebssystem mit Sicherungseinrichtungen zu schreiben.

Problem: Neue Programme. Das Verfahren von UNIX eignet sich nicht für alle Aspekte der Sicherheit unter MS-DOS. Die Probleme fangen damit an, daß ständig neue Programme ins System kommen, sei es durch Kompilierung von Quelltexten, Übertragung per DFÜ oder Laden (Starten!) von Diskette. Diese Programme sind in keiner Rechteliste enthalten, müssen aber ebenfalls geschützt und überprüft werden (z.B. generelles Verbot für Programmstarts von Floppylaufwerken).

Problem: Beschränkung nach Aufgaben. Überdies ist es interessant, nicht nur die Berechtigung des hinter jedem Programm stehenden Anwenders zu überprüfen, sondern die des Programms selbst. Der *Capability*-Ansatz geht davon aus, daß jeder Programmtyp bestimmte Rechte hat. Beispielsweise ist der Zugriff eines Compilers auf eine ausführbare Datei völlig legitim, was sich von einem Textverarbeitungsprogramm nicht sagen läßt. Unter UNIX geht man davon aus, daß alle im System befindlichen Programme das tun, was man von ihnen erwartet. Falls das aber nicht zutrifft, weil z.B. ein Trojaner an die Stelle eines privilegierten Programms getreten ist, darf dieser auf jede beliebige Datei ungehindert zugreifen. Auf Programme darf sich deshalb nur verlassen werden, wenn deren Integrität und bestimmungsgemäße Funktion gewährleistet ist.

Beispiel "Flushot"
`Flushot` (Autor: Ross Greenberg) ist ein Programm für MS-DOS-Rechner, das Schutzmechanismen auf Dateiebene realisiert (Tab. 4.17). `Flushot` führt folgende Neuerungen gegenüber dem UNIX-Konzept ein:

- Verwendung von Jokerzeichen in der Dateispezifikation (z.B. Schreibschutz für `*.exe`),
- Definition von Ausnahmen von Schutzoptionen ('x' und 'e'),
- Definition von Pflichten ('c'),
- Speicherung der Rechte unabhängig von der Dateiverwaltung durch das Betriebssystem.

Option	Objekt	Bedeutung	
P	Datei	write **p**rotect	Schreibschutz
R	Datei	**r**ead protect	Leseschutz
E	Datei	**e**xclude from protection	von Dateischutz ausnehmen
X	Programm	e**x**eption	von besonderen Schutzmaßnahmen ausnehmen
T	Programm	valid **T**SR	zulässiges TSR-Programm
C	Programm	compute **c**hecksum	Prüfsumme berechnen

Tabelle 4.17: Schutzmaßnahmen unter `Flushot`

Weil die Optionen 'P', 'R' und 'E' nicht das aufrufende Programm berücksichtigen, kommt es häufig zu Fehlalarmen. So meldet sich `Flushot` beispielsweise, wenn der Schreibzugriff auf ausführbare Dateien verboten ist und ein Compiler ein neues Programm schreiben will. Umgekehrt bewirkt die Option 'X', daß ein Programm gänzlich von der Überwachung ausgeschlossen wird. Aus diesen Gründen berücksichtigt `AVWatchF` stets das aufrufende Programm, die angeforderte Operation und die zu bearbeitende Datei.

Aufbau und Zugriff auf die Rechtedatei. Die Rechtedatei `f_rights.raw` enthält pro Zeile einen Eintrag in Klartext mit folgendem Aufbau:

```
Recht   ::= <Subjekt> <Rechte> <Objekt>
Subjekt ::= Programmname
Objekt  ::= Programmname
Rechte  ::= {Option(en)}
Option  ::= s|f|i|t|r|w|x
```

Die Reihenfolge der Rechte spielt keine Rolle, wohl aber die Groß- und Kleinschreibung. Tabelle 4.18 stellt die von `AVWatchF` realisierten Optionen und deren Codierung vor.

Option	Bit	Objekt	Bedeutung	
x	0	Programm	e**x**ecute allowed	Ausführen zulässig
w	1	Datei	**w**rite allowed	Schreiben zulässig
r	2	Datei	**r**ead allowed	Lesen zulässig
t	3	Programm	valid **t**SR	zulässiges TSR-Programm
i	13	Programm	check **i**ntegrity	Integrität überprüfen (durch Funktionen aus `AVWatchI`)
f	14	Zulassung	log if **f**ail	Log falls unzulässig schreiben
s	15	Zulassung	log if **s**uccess	Log falls zulässig schreiben

Tabelle 4.18: Schutzmaßnahmen unter `AVWatchF`

Die kodierte Rechtedatei `f_rights.lst` besteht aus Einträgen fester Länge, um die Zugriffszeit gering zu halten. Datensätze variabler Länge erfordern zwar weniger Speicherplatz, sind aber aus Geschwindigkeitsgründen für unsere Zwecke ungeeignet. Hier zeigt sich ein Prinzip der Systemprogrammierung: Geschwindigkeit kann oft durch Speicherplatz erkauft werden und umgekehrt. Der Aufbau eines Eintrags in `f_rights.lst` ist wie folgt:

```
struct T_FR_ENTRY
{ char subject[65];
  WORD rights;
  char object[65];
};
```

Die Umsetzung der Textdatei `f_rights.raw` in die kodierte Datei `f_rights.lst` übernimmt die Konvertierungsroutine `compile_f_rights` in `AVConfig`.

Interpretation der Rechte. Unter UNIX ist für jede Datei genau ein Rechtetupel definiert. Anders die Situation bei `AVWatchI`. Ein Eintrag in der Rechtedatei kann mehrere Dateien betreffen, mehrere Einträge können auf eine Datei passen (und sich widersprechen) und es muß nicht für jede Datei ein passender Eintrag existieren.

Für den Fall, daß kein Eintrag zutrifft, gibt es zwei mögliche Verfahren.

1. "Es ist alles erlaubt, was nicht verboten ist"
   Dieses Verfahren erfordert vom Systemadministrator einige Aufmerksamkeit, weil sein Rechner quasi als Weglaßwert über keine Schutzmaßnahmen verfügt (Discretionary Access Control).
2. "Es ist alles verboten, was nicht erlaubt ist"
   Philosophie: Was der Systemverwalter vergessen hat, den Anwendern zu erlauben,

kann keinen Schaden anrichten. Bei sicheren Systemen im Sinne des Orange Book ist dieses Verfahren Pflicht (Mandatory Access Control).

Treffen mehrere Rechteeinträge zu, wird die Sache schon komplizierter. Ein Beispiel verdeutlicht das:

```
1. *.* rx *.EXE           % niemand darf EXE-Dateien schreiben
2. C:\TC\TC.EXE rwx *.EXE  % Ausnahme: Turbo-C muss und darf
```

Angenommen, `C:\TC\TC.EXE` (Turbo-C) kompiliert ein Programm und möchte die Zieldatei schreiben. `AVWatchF` fängt die Anforderung ab und findet bei der Überprüfung zwei passende Einträge. Ein Mensch würde folgendermaßen entscheiden: Der erste Eintrag paßt auf alle Programme, der zweite nur auf ein einziges Programm. Der zweite Eintrag stellt somit die Ausnahme dar und *paßt besser*. Das führt uns zu einer Vergleichsfunktion, die ein Maß für den Grad der Übereinstimmung zurückliefert, um den am besten passenden Eintrag auswählen zu können.

`cmp_fspec` vergleicht im `FM_LAX`-Modus nur die Teile des Dateinamens mit der Maske, die in der Maske vorkommen (Tab. 4.19). Laufwerk und Verzeichnis können fehlen und tragen dann nicht zum Vergleichsergebnis bei. Der Dateinamen wird immer mit `cmp_fname` überprüft und bewertet. Wir wollen so verfahren, daß Übereinstimmungen bei Laufwerk und Verzeichnis eine höhere Priorität als bei Dateinamen haben. Für jeden passenden Teil, Laufwerk oder Verzeichnis, gibt es $256_{10} = 0100_{16}$ Punkte, die zu den Punkten von `cmp_fname` addiert werden. Der Rückgabewert von `cmp_fspec` enthält demnach im höherwertigen Byte die Anzahl der Laufwerks/Pfadtreffer und im niederwertigen Byte die Anzahl der Treffer im Dateinamen. Maske Nr.1 aus dem Beispiel brächte es beim Subjekt auf die Wertung 0000_{16}, während Maske Nr.2 mit 0205_{16} den Zuschlag erhält. Die Funktion `get_rights` addiert die Wertungen von Subjekt und Objekt, um zu einem Ergebnis zu gelangen (s. Beschreibung im Anhang A.4).

Funktion "Compare Filespecifications":
`WORD cmp_fspec (char mode, char f_spec[], char mask[])`

Aufrufparameter:

`mode`	Überprüfung: {0, `FM_STRICT`: vollst. Spezifikation; 1, `FM_LAX`: nur in `mask` vorhandene Namensteile}
`f_spec`	Dateispezifikation
`mask`	Maske (kann Jokerzeichen enthalten)

Seiteneffekte:
keine

Rückgabewert:
niederwertiges Byte: s. `cmp_spec`; höherwertiges Byte: Anzahl passender Teile (Laufwerk und Pfad)

Tabelle 4.19: `cmp_fspec`: Vergleiche Dateispezifikationen

Funktionsbeschreibung. Zunächst sind die üblichen Umbauten an `Std_TSR` notwendig, die durch die Kommentare ausreichend erläutert werden.

```
/* einfuegen in "Defines"                                          */
#define F_DEFAULT       ON
#define F_INTERACT      YES

/* Anzahl der zu ueberwachenden Funktionen (Groesse SET, s. ISR_21)*/
#define M_SET           2
#define PRG_NAME        "AVWatch (F)"
#define PRG_ID          ('f' << 8)

/* einfuegen in "globale und externe Variablen"                    */
char    f_log[65];                      /* vollst. Name Logdatei   */
char    f_rights[65];                   /* vollst. Name Rechtedatei*/

/* einfuegen in INIT                                               */
  /* erzeuge vollst. Dateinamen                                    */
  add_path (argv[0], "f_rights.lst", f_rights);
  add_path (argv[0], "avwatch.log", f_log);
```

Klassifizierung der Zugriffe. Analog zu `AVWatchP` wird die aufgerufene Funktion zunächst mit Hilfe der Informationen in `set` klassifiziert. Ist `func` nicht in `set` enthalten, wird der Aufruf nach Prüfung auf Serviceanforderungen normal ausgeführt. Sonst enthält `n_set` die Nummer des zuständigen Eintrags. Nächster Schritt ist die Bestimmung von Subjekt (aufrufendes Programm), Operation (erforderliche Rechte) und Objekt (Datei).

```
/* einfuegen in "Typedefs"                                         */
struct T_SET
{ BYTE  func;
  WORD  mode;
  WORD  rights;
  WORD  errno;
};

/* einfuegen in ISR_21                                             */
  static struct T_SET set[M_SET] =      /* Pruefinformation        */
  { {0x4B, (0x00 << 11) | (0x00 << 5) | 0x00, 0x0001, 0x0002},
    {0x6C, (0x00 << 11) | (0x02 << 5) | 0x00, 0x0000, 0x0005}
  };
```

```
static char *rights_text[] =
{ "execute-" ,
  "write-",
  "read-"
};
static char object[65];              /* Objekt (Datei)          */
static char subject[65];             /* Subjekt (Programm)      */
static char msg[150];                /* Meldungspuffer          */
struct T_MCB far *mcb_ptr;           /* fuer GET_OWNER_MCB      */
WORD rights;                         /* Rechte aus F_RIGHTS.LST */
WORD req;                            /* erforderliche Rechte    */
int  i, j;                           /* Indizes in String       */
int n_set;                           /* Pruefeintragsnummer     */
BYTE func;                           /* Funktionsnummer         */
BYTE far *text;                      /* FAR-Zeiger auf Objekt   */
char decide;                         /* Entscheidungsflag       */

/* suche Pruefeintrag                                           */
func  = ax >> 8;
n_set = 0;
while ((n_set < M_SET) && (set[n_set].func != func))
{ n_set++;
};
```

Das Modus-Wort in `set` ist in drei Bitfelder aufgespalten:

Bits	Funktion	Bedeutung
15 – 11	Objekt-Modus	0: Handle-Aufruf (`DS:DX` oder `DS:SI` zeigt auf Objekt)
10 – 5	Rechte-Modus	0: in `rights`; 1: Handle Open; 2: ext. Handle Open
4 – 0	Fehler-Modus	0: Handle; 1: FCB Typ 1; 2: FCB Typ 2

Tabelle 4.20: Aufbau des Modus-Wortes in `set` (`isr_21`)

Zur Extraktion der einzelnen Modi stehen drei Makros zur Verfügung:

```
/* aus dem Kopf von "AVWatchF.C"                                */
#define GET_E_MODE(x) (x & 0x1F)        /* Fehler-Modus         */
#define GET_O_MODE(x) (x >> 11)         /* Objekt-Modus         */
#define GET_R_MODE(x) ((x >> 5) & 0x3F) /* Rechte-Modus         */
```

Bestimmung Subjekt. Die Kombination der Aufrufe von `get_owner_mcb` und `get_mcb_name` füllt `subject` mit dem Namen des aufrufenden Programms.

```
rights = 0;                          /* kein Eintrag->kein Log. */
if (n_set < M_SET)                   /* Eintrag gefunden        */
{ /* bestimme Subjekt (aufrufendes Programm)                    */
  decide = -1;
  if (get_owner_mcb (ret_adr, &mcb_ptr) != 0xFFFF)
  { get_mcb_name (mcb_ptr, subject);
```

Dabei ergeben sich einige durch das Betriebssystem verursachte Probleme. MS-DOS weiß intern nicht, welches Programm es gerade ausführt. Deshalb müssen wir umständlich über Rücksprungadresse, MCB-Kette und Environment den Aufrufer einer Funktion bestimmen. Leider gibt es außerdem noch Programme, die ihr Environment manipulieren und damit die Ermittlung des Programmnamens erschweren oder verhindern.

Bei `command.com` beispielsweise ergibt die Feststellung des Programmnamens stets `command.com`, d.h. ohne Laufwerk und Unterverzeichnis. Eine Normalisierung wird deshalb generell nicht durchgeführt, da dies zu falschen Schlüssen führen könnte.

Fatal wird die Situation bei Programmen, die an MS-DOS komplett vorbeiwirtschaften. Z.B. lagert der alternative (und gut gelungene) Kommandointerpreter `4dos` Teile des Programmcodes in den Erweiterungsspeicher aus. Zur Ausführung einer Funktion wird der Programmcode oder Teile davon in den normalen Speicher zurückverlagert, allerdings ohne daß dazu konventionelle Funktionen zur Speicherreservierung verwendet werden. Das führt dazu, daß bei Kernel-Aufrufen innerhalb dieses Codes die Rücksprungadresse scheinbar im freien Speicher liegt — die Ermittlung des Verursachers ist somit unmöglich.

Unter Multitasking-Betriebssystemen ist diese Art von Programmierung gar nicht möglich, weil kein Programm annehmen darf, daß es kontinuierlich abläuft und alle Ressourcen für sich allein beanspruchen kann. Bei den angeführten Problemen ist die Suche nach alternativen Methoden oder die Verwendung von "anständiger" Software angesagt. Wir bleiben der Einfachheit halber bei Programmen, die sich an die durch MS-DOS vorgegebenen Funktionen halten.

Bestimmung Objekt. Der Dateiname wird entweder als String gemäß der "C"-Konvention übergeben (Handle-Aufrufe) oder steht in einem FCB. Bei den Handle-Aufrufen zeigt das Registerpaar `DS:DX` auf den Dateinamen. Eine Ausnahme macht die seit Version 4.0 verfügbare Funktion $6C_{16}$ "Extended Open File" mit dem Registerpaar `DS:SI`, was auch eher der Philosophie der Standardregister entspricht. Bei den FCB-Aufrufen zeigt `DS:DX` auf einen "normal" oder "extended" FCB. Im normalen FCB steht die Laufwerksnummer an Offset 00_{16}, unmittelbar gefolgt vom Dateinamen im FCB-Format. Der erweiterte FCB ist durch den Wert FF_{16} an Offset 00_{16} gekennzeichnet. Laufwerk und Name beginnen in diesem Fall bei Offset 07_{16}, also um 7 Bytes nach hinten verschoben.

Nach dem Öffnen einer Datei mit UNIX-ähnlichen Funktionen erfolgen weitere Aufrufe wie Lesen oder Schreiben über Angabe des zugehörigen Handles. Um diese Funktionsaufrufe kontrollieren zu können, müßte das Kontrollprogramm entweder selbst eine Liste der Handles mitführen oder auf interne, undokumentierte Funktionen von MS-DOS zurückgreifen. Von letzterer Methode ist dringend abzuraten, da sich die relevanten internen Datenstrukturen (System File Table und Job File Table) schon von Release zu Release ändern. Die erste Methode hingegen erfordert hohen Programmieraufwand. Aus diesen Gründen beschränken wir uns auf die Kontrolle des ersten Zugriffs, das Öffnen der Datei, das ja in jedem Fall erfolgen muß.

FCB-Aufrufe unterscheiden keine Pfade[12], sondern beziehen sich stets auf das bei Offset 00_{16} bzw. 07_{16} angegebene Laufwerk und das aktuelle Verzeichnis darauf. Deshalb muß der Dateiname noch um die Laufwerksangabe ergänzt werden. Eine Normalisierung ist generell notwendig, um Dateinamen miteinander vergleichen zu können. Zu beachten ist außerdem, daß die oben angesprochenen Registerpaare FAR-Zeiger bilden, wir aber im `TINY`-Modell arbeiten. Die "C"-Bibliotheksfunktionen erwarten die Übergabe von NEAR-Zeigern auf Daten und Programmcode und versagen bei Mißachtung

[12] CP/M kannte kein hierarchisches Dateisystem, sondern war "flach" organisiert.

dieses Sachverhalts ebenso natürlich wie kläglich. Daher kopieren wir vor der Weiterverarbeitung die FAR-Strings in NEAR-Strings um.

```
/* bestimme Objekt                                          */
switch (GET_O_MODE (set[n_set].mode))
{ case 0x00:                        /* Handle (DS:DX/SI)?   */
    text = MAKE_FARPTR (ds, (func != 0x6C) ? dx : si);
    i = 0;                          /* konv. FAR in NEAR-String*/
    while ((object[i] = *(text++)) != '\0')
    { i++;
    };
    break;
  case 0x01:                        /* FCB?                 */
    text = MAKE_FARPTR (ds, dx);
    if (*text == 0xFF)              /* erweiterter FCB?     */
    { text += 0x07;
    };
    if ((i = *(text++)) == 0)       /* aktuelles Laufwerk?  */
    { i = 1 + xgetdisk ();
    };
    object[0] = '@' + i;            /* Laufwerksangabe->OBJECT */
    object[1] = ':';
    j = 2;                          /* konv. FAR in NEAR String*/
    for (i = 0; i < 11; i++)
    { if (text[i] != ' ')
      { if (i == 8)
        { object[j++] = '.';
        };
        object[j++] = text[i];
      };
    };
    object[j] = '\0';
    break;
};
norm_path2 (object);
```

Bestimmung Operation. Die erforderlichen Rechte ergeben sich entweder direkt aus `set` oder indirekt aus der Subfunktionsnummer. Bei den Funktionen $3D_{16}$ "Open file" und $6C_{16}$ "Extended Open File" bestimmt `AL` bzw. `BX` den Zugriffsmodus. Relevant sind in beiden Fällen die Bits 0 bis 2, die als Index in die Tabelle `table` dienen:

```
/* aus der Variablendeklaration in ISR_21                   */
static WORD table[] =               /* erf. Rechte fuer OPEN */
{ 0x0004, 0x0002, 0x0006
};

/* Fortsetzung des Codes                                    */
  /* bestimme erforderliche Rechte                          */
  switch (GET_R_MODE (set[n_set].mode))
  { case 0x00:                      /* implizit             */
      rights = set[n_set].rights;
      break;
    case 0x01:                      /* "Handle Open"        */
      bx = ax;
    case 0x02:                      /* "Ext. Handle Open"   */
      rights = table[bx & 0x03];
      break;
  };
  req = rights;
```

```
  decide = get_rights (subject, &rights, object, F_DEFAULT);
};
```

Bewertung des Zugriffs. Nun haben wir alle Parameter für den Aufruf von `get_rights` zusammen. Wird kein Eintrag gefunden oder passen mehrere Einträge gleich gut, entscheidet der Wert `F_DEFAULT` über die Zulässigkeit der Operation. Das Ergebnis der Überprüfung wird je nach Festlegung des Makros `F_INTERACT` analog zum Verfahren in `AVWatchI` gehandhabt. Ist die Operation zulässig, wird die ISR mit dem Rückgabewert $FFFF_{16}$ verlassen (Aufruf der Original-ISR).

```
/* einfuegen in "Defines"                                          */
#define FR_LOGFAIL      0x4000          /* Log falls unzulaessig   */
#define FR_LOGSUCC      0x8000          /* Log falls zulaessig     */

    if (decide < 0)
    { decide = F_DEFAULT;
    };

    /* schreibe Logeintrag (falls spezifiziert; s. "Logging")      */
    if (((rights & FR_LOGSUCC) && (decide == 1)) ||
        ((rights & FR_LOGFAIL) && (decide == 0)))
    { add2log (f_log, subject, (DWORD)ax << 16, object);
    };

    if (decide == 1)
    { return (0xFFFF);
    };

    if (F_INTERACT)
    { strcpy (msg, subject);            /* Nachricht erzeugen      */
      strcat (msg, " -");
      for (i = 0; i < 3; i++)
      { if (req & 0x0001)
        { strcat (msg, rights_text[i]);
        };
        req >>= 1;
      };
      strcat (msg, "> ");               /* Benutzer fragen         */
      strcat (msg, object);
      strcat (msg, ". ");
      strcat (msg, "Allow 'y'es / no");
      if ((message (0x4F, msg) & 0xFF) == 'y')
      { return (0xFFFF);
      };
    };
```

Im Fehlerfall ist der Rückgabewert gleich der zu simulierenden Fehlernummer (Abbruch des Aufrufs).

```
    /* bestimme zu simulierenden Fehler                            */
    switch (GET_E_MODE (set[n_set].mode))
    { case 0x00:                        /* Carry = 1, AX = errno   */
      case 0x01:                        /* FCB1 (AL = 0xFF)        */
      case 0x02:                        /* FCB2 (AL = 0x01)        */
        return (set[n_set].errno);
    };
  };
```

Logging. `AVWatchF` leistet in der aktuellen Entwicklungsstufe eigentlich schon alles, was für den Schutz vor Manipulationsversuchen durch Softwareanomalien und

Anwender notwendig ist. Beim Einsatz von APCs in Rechenzentren oder Betrieben ist eine ständige Sichtkontrolle aus Gründen des Aufwands und anderen Gesichtspunkten nicht möglich. Abhilfe schafft die automatische Protokollierung von bestimmten Systemoperationen, mit deren Hilfe eine nachträgliche Kontrolle erfolgen kann.

Noch einmal zurück zur Flugdatenschreiber-Analogie.

Selbst bei einem Crash des Flugzeugs kann nachträglich noch ermittelt werden, was zum Unglück führte. Dafür muß sichergestellt sein, daß der Flugdatenschreiber so gepanzert ist, daß dieser den Aufprall und Feuer übersteht. Eine Untersuchungskommission kann durch Auswertung der Daten die Absturzursache feststellen, Verantwortung zuweisen sowie dazu beitragen, die Flugsysteme zu verbessern.

Selbst bei einer Verseuchung des Computers kann nachträglich noch ermittelt werden, was zur Infizierung führte. Dafür muß sichergestellt sein, daß die Protokolldatei so vor Zugriff geschützt ist, daß diese Manipulationsversuche des Virus oder des Benutzers übersteht. Der Systemverwalter kann durch Auswertung der Daten den Infektionsweg feststellen, Verantwortung zuweisen sowie dazu beitragen, die Systemsicherheit zu verbessern.

Jeder Logeintrag besteht aus einer Zeitmarke (Datum, Uhrzeit) und den Parametern der beanstandeten Operation. Normalerweise gehört die Benutzerkennung unbedingt dazu; aber da MS-DOS so etwas nicht unterstützt, ist diese Angabe hinfällig. Wird Auskunft über den Benutzer gewünscht, sind eigene Schutzprogramme vorzusehen, die vom Anwender vor der Freigabe des Rechners Angaben zur Identität verlangen.

Indirekt kann das Gleiche über die Einführung von Regeln erreicht werden, die jedem Anwender vorschreiben, daß Name und Zeitraum der Rechnerbenutzung in eine Liste einzutragen sind. Über die Zeitmarke in der Protokolldatei kann dann manuell auf den Verursacher geschlossen werden. Diese Methode hängt natürlich vom guten Willen der Benutzer ab, den unregelmäßige Kontrollen stärken können (Entzug der Benutzungserlaubnis androhen etc.). Da die Zeitmarke eine wesentliche Rolle bei der Identifikation des Verursachers spielt, müssen die Funktionen zum Ändern von Systemdatum und -zeit gegen unbefugte Benutzung gesichert sein (s.a. "`AVWatchG`").

Zur Kennzeichnung der Operation eigenen sich Subjekt, Operation und Objekt. "Operation" sollte nicht nur in Lesen, Schreiben, Ausführen etc. untergliedert sein, sondern genau die Betriebssystemfunktion angeben. Unter MS-DOS gehören dazu die Nummern von Interrupt, Funktion, Subfunktionen oder statt dessen weitere Parameter.

Add Entry to Logfile (add2log) fügt einen Logeintrag an die Logdatei `f_name` an (Tab. 4.21). Dieser hat die Form

```
struct T_LOG_ENTRY
{ struct T_DATE date;
  struct T_TIME time;
  char   subject[65];
  DWORD  func_code;
  char   object[65];
};
```

```
struct T_DATE
{ int  year;
  char day;
  char month;
};

struct T_TIME
{ unsigned char minute;
  unsigned char hour;
  unsigned char centi;                  /* 100stel Sekunden        */
  unsigned char second;
};
```

Datum und Zeit werden von `add2log` aktuell ermittelt und eingesetzt. Die Belegung von `func_code` sollte so erfolgen, daß die Operation möglichst genau spezifiziert wird. Das Reportprogramm `LogRep` auf der Begleitdiskette erwartet folgenden Aufbau (vom höchstwertigen zum niederstwertigen Byte): Interruptnummer (je nach ISR), Funktion (meist in `AH`), Subfunktion (meist `AL`) und Subsubfunktion.

```
int add2log (char f_name[], char subject[],
             DWORD func_code, char object[])
{ struct T_LOG_ENTRY log;               /* Logeintrag              */
  int handle;                           /* Handle Logdatei         */
  char err;                             /* Fehlerstatus            */

  if ((handle = open (f_name, O_WRONLY | O_APPEND | O_BINARY)) == -1)
  { message (0xCF, "Logfile open error");
    return (-1);
  };

  err = 0;
  xgetdate (&log.date);                 /* baue Logeintrag auf     */
  xgettime (&log.time);
  strcpy   (log.subject, subject);
  log.func_code = func_code;
  strcpy   (log.object, object);
  /* schreibe Logeintrag                                          */
  if (write (handle, &log, sizeof (struct T_LOG_ENTRY)) !=
    sizeof (struct T_LOG_ENTRY))
  { message (0xCF, "Logfile write error");
    err = -2;
  };
  close (handle);

  return (err);
};
```

Hinweis. Die Logdatei muß vor dem ersten Gebrauch mit der Eingabe von "`copy con avwatch.log`" gefolgt von Ctrl Z in dem Verzeichnis angelegt werden, in dem sich auch die Watcher befinden. Um die Logdaten vor unbefugtem Zugriff zu schützen, sollten die Attribute `hidden` und `system` gesetzt werden; das Ändern von Dateiattributen sollte nicht zulässig sein.

Log Report (LogRep). Der Reportgenerator konvertiert die Logdatei in eine für Menschen verständliche Form, indem statt der rohen Parametern der Funktionsaufrufe symbolische Namen verwendet werden. So wird die knappe Loginformation "$213D0000_{16}$" zum aussagefähigen Text "Open File (Read Access, Compatibility Mo-

Funktion "Add Entry to Logfile":
`int add2log (char f_name[], char subject[], DWORD func_code, char object[])`

Aufrufparameter:

`f_name`	vollst. Name der Logdatei
`subject`	Subjekt (aufrufendes Programm)
`func_code`	Operation (Interruptnummer, Funktion, Subfunktion, Subsubfunktion)
`object`	Objekt (Dateiname)

Seiteneffekte:
An die Logdatei wird ein Eintrag angehängt

Rückgabewert:
0: kein Fehler; <0: Fehler

Tabelle 4.21: `add2log`: Füge Eintrag an Logdatei an

de, Child Inherits Handle)". Das auf der Diskette enthaltene Programm `LogRep` ist ein Beispiel für eine einfache Textausgabe der Logdatei ohne weitere Extras.

Ein komfortableres Reportprogramm sollte die Suche nach bestimmten Zeiträumen, Operationen, Parametern etc. unterstützen, damit das Datenmaterial auf sicherheitskritische, relevante Vorgänge reduziert werden kann. Praktisch ist ein Stapelprogramm, das z.B. automatisch am Ende des Tages alle Logdateien auf suspekte Manipulationen untersucht und einen Sicherheitsbericht zusammenstellt.

4.5.9 Mögliche Erweiterungen

Schaut man sich bei kommerzieller Software zur Abwehr von Computerviren um, kann man eine Reihe Funktionen entdecken, die wir noch nicht angesprochen haben und um deren Relevanz man sich streiten kann. Der Autor hat insgesamt mehr als 25 Antivirusprogramme, die über den Server `trickle@ds0rus1i.bitnet` (s.a. B.4 "TRICKLE-Server") vertrieben wurden oder werden, auf funktionale Eigenschaften untersucht. Die folgende Aufstellung kann uns vielleicht noch einen Hinweis auf sinnvolle Erweiterungen geben. Schon angesprochene Konzepte wurden aus Gründen der Übersichtlichkeit weggelassen.

- Kontrolle von ANSI-Sequenzen an den Bildschirm-/Tastaturtreiber
- Hardwaremäßiger Bootschutz
- Paßwortabfrage beim Bootvorgang durch Gerätetreiber
- "Abschließen" des Computers per Paßwort für kurze Abwesenheit
- Dateiverschlüsselung auf Soft- und Hardwarebasis

- Löschen von Dateien durch Überschreiben bis Clustergrenze
- Paßwortüberprüfung (Prüfung der Qualität)
- Shells für die Überprüfung von gepackten Programmen

Manche dieser Optionen haben nicht unmittelbar mit der Abwehr von Computerviren zu tun, sind aber trotzdem nützlich und unterstützen die Systemsicherheit.

ANSI-Sequenzen. Ein "trojanischer Text" könnte über ANSI-Sequenzen Tasten mit Befehlen belegen, deren Ausführung unerwünschte Effekte nach sich zieht. Zur Abwehr ist die Kontrolle aller Funktionen erforderlich, die Zeichen auf den Bildschirm ausgegeben und dazu den ANSI-Treiber benutzen. Allerdings wurde noch keine Softwareanomalie dieser Art bekannt.

CMOS-RAM. Die Kontrolle des sog. CMOS-Speichers ist ein nicht unbedingt notwendiges Extra mancher Watcher. Der Chip für die Echtzeituhr, den jeder AT enthält, ist in besonders stromsparender CMOS-Technologie ausgeführt. Dies ermöglicht den kontinuierlicher Betrieb auch bei ausgeschaltetem Rechner mit Hilfe eines Akkus. Der Uhrenbaustein besitzt neben den Timerfunktionen einen 64 Byte großen Speicher, der das Abschalten der Versorgungsspannung ohne Datenverlust übersteht. Über BIOS-Funktionen oder direkte Portzugriffe, nicht aber über den normalen Adreß- und Datenbus, kann der Speicher gelesen und beschrieben werden.

Das BIOS verwendet Teile des CMOS-RAMs für Konfigurationsdaten wie Anzahl und Typ der Festplatten und Diskettenlaufwerke, Größe des Arbeitsspeichers und einige Angaben mehr. Über das sog. *Setup*-Programm, das sich meist beim Bootvorgang über eine spezielle Tastenkombination aufrufen läßt, können die Parameter durch den Benutzer verändert werden. Ein Virus, das die Systemkonfiguration verändern will, kann dazu (abfangbare) BIOS-Funktionen aufrufen oder Portzugriffe verwenden, die sich der Kontrolle entziehen. Die einzige Manipulationskontrolle besteht im regelmäßigen Vergleich (z.B. über den Timer-Interrupt $1C_{16}$) des CMOS-RAMs mit einer Kopie des Inhalts oder einer Prüfsumme. Falls das Virus die vom Setup-Programm generierte und vom BIOS beim Urladevorgang überprüfte interne Prüfsumme nicht verändert, gibt das Betriebssystem beim nächsten Hochfahren des Systems eine Fehlermeldung aus. Bei "korrekter Manipulation" arbeitet das BIOS u.U. mit falschen Laufwerksparametern und kann bei Schreibzugriffen Daten zerstören. Es ist aber noch kein Virus bekannt (Ende 1991), das sich diese Manipulationsmethode zu nutze macht.

Es ist eine Legende, daß manche Viren ihren Code im Konfigurations-RAM ablegen und von dort irgendwie zum Ablauf bringen. Allein schon, weil der CMOS-Speicher nicht Bestandteil des Adreßraums ist, kann diese Behauptung nicht stimmen. Möglich wäre es allerdings, daß ein Virus diesen Speicher zur Ablage von Hilfsdaten wie z.B. einen Infektionszähler verwendet.

Hardwaremaßnahmen. Mit Hardwaremaßnahmen werden uns nicht näher beschäftigen, weil unser Schwerpunkt auf der Systemprogrammierung liegt. Trotzdem ein paar Anmerkungen zu diesem Thema. In Zukunft wird auf in der Hardware implementierte Schutzeinrichtungen nicht mehr verzichtet werden können. Bestes Argument

sind die momentan weltweit eingesetzten IBM- oder Apple-PCs, die entweder keine Speicherschutzmechanismen besitzen oder nutzen (s. Real Mode/Protected Mode ab dem 80286-Prozessor). Solange jedes Programm auf jede Speicherstelle des Systems nach belieben zugreifen kann, wird es keinen Schutz vor Unannehmlichkeiten wie Computerviren geben.

Aber auch die Betriebssysteme sind noch stark entwicklungsfähig. Das "Orange Book" und die IT-Sicherheitskriterien führen zusammen mit dem Wunsch der Kunden nach mehr Sicherheit und den Bestrebungen der Hersteller, diese Auflagen zu erfüllen, zumindest für Großrechner in die richtige Richtung.

4.5.10 AVConfig

In `AVConfig` sind mehrere Hilfsprogramme zusammengefaßt:

- Übersetzung der Rechtedatei `f_rights.raw` (Text) in die kodierte Form `f_rights.lst` für `AVWatchF` (`compile_f_rights`),
- Ermittlung der Partitionsdaten und Übersetzung der Rechtedatei `t_rights.lst` sowie Übertragung der Daten an `AVWatchP` (`im_i_table`),
- Deinstallierung von residenten Programme (`im_deinst`),
- Anfügen/Entfernen des Programmsiegels für die Überprüfung durch `chk_seal` (`mod_seal`).

Die Definitionen und globalen Variablen am Anfang der Datei sind vom Prinzip her allesamt bekannt. `main` setzt den Zeiger `msg_ptr` auf `msg` und prüft, ob beim Aufruf von `AVConfig` Parameter angegeben wurden. Ist das der Fall, arbeitet `AVConfig` im Batch-Modus und verzweigt zur Funktion `batch`. Diese Option ist für die Initialisierung von `AVWatchP` wichtig, da nach der Installierung noch die Partitionsdaten und -rechte automatisch übertragen werden müssen. Liegen keine Parameter vor, bietet die Funktion `interactive` dem Anwender ein Menü an und wartet auf eine Eingabe.

```
/* Defines                                                    */
#define BUF_SIZE        512
#define CRC_START       0x0000
#define M_DRIVE         10              /* max. Anzahl log. Laufw. */
#define POLYNOMIAL      0xA001
#define PWD             0x123456781

/* globale und externe Variablen                              */
struct T_I_MESSAGE msg, far *msg_ptr;   /* Nachricht, Zeiger auf N.*/
struct T_I_TABLE i_table;               /* fuer IM_I_TABLE         */
```

Sowohl im Batch- als auch im interaktiven Modus übernimmt `switch_function` die weitere Verteilung an die einzelnen Funktionen. `batch` ruft `switch_function` mit dem ersten Buchstaben des ersten Kommandozeilenarguments auf, `interactive` übergibt die vom Benutzer erfragte Nummer eines Menüpunktes.

```
int switch_function (int argc, char *argv[], char choice)
{ char  raw[65], cooked[65];           /* Namen der Rechtelisten  */
  int   result;                        /* Ergebnis der Aufrufe    */

  printf ("\n");
  switch (choice)
  { case '1': case 'f':                /* kompiliere Dateirechte  */
      add_path (argv[0], "f_rights.raw", raw);
      add_path (argv[0], "f_rights.lst", cooked);
      result = compile_f_rights (raw, cooked);
      break;
    case '2': case 'p':                /* uebertrage Part.-Daten  */
      add_path (argv[0], "p_rights.lst", raw);
      result = im_i_table (raw);
      break;
    case '3': case 'd':                /* deinstalliere Watcher   */
      result = im_deinst ();
      break;
    case '4': case 's':                /* bearbeite Dateisiegel   */
      result = mod_seal ();
      break;
    default:
      fprintf (stderr, "Unknown option \"%c\"\n", choice);
      result = -1;
  };
  printf ("\n");
  return (result);
};
```

compile_f_rights wandelt die Rechtetripel in der Textdatei `f_rights.raw` in ein maschinenfreundlicheres Format mit `f_rights.lst` als Zieldatei um. Dazu wird die Rohdatei zeilenweise in die Struktur `fr_entry` (**f**ile **r**ights entry) und in den String `rights_str` eingelesen. Während Subjekt und Objekt schon in der endgültigen Form vorliegen, muß `rights_str` noch mit `tokenise` in `fr_entry.rights` konvertiert werden. Der so vervollständigte Eintrag wird in die Zieldatei geschrieben.

```
int compile_f_rights (char source[65], char dest[65])
{ FILE  *in, *out;                     /* Roh-/Zieldatei          */
  struct T_FR_ENTRY fr_entry;          /* Dateirechteeintrag      */
  char  rights_str[17];

  printf ("Compiling \"%s\" -> \"%s\"\n", source, dest);
  if ((in = fopen (source, "r")) == NULL)
  { fprintf (stderr, "Couldn't open file rights file\n");
    return (-1);
  };
  if ((out = fopen (dest, "wb")) == NULL)
  { fclose (in);
    fprintf (stderr, "Couldn't open destination file\n");
    return (-2);
  };

  while (fscanf (in, "%64s %16s %64s",
    fr_entry.subject, rights_str, fr_entry.object) == 3)
  { /* konvertiere Rechte-String in Rechte-Wort                    */
    fr_entry.rights = tokenise (rights_str, "sfi         trwx");
    if (fwrite (&fr_entry, sizeof (struct T_FR_ENTRY), 1, out) != 1)
    { fprintf (stderr, "Write error\n");
      return (-3);
```

```
    };
  };
  fclose (in);
  fclose (out);
  return (0);
};
```

im_i_table dient zum "Beladen" von `AVWatchP` mit den Partitionsdaten und -rechten. Die Funktion beginnt mit einem Trick, der ausnutzt, daß Disketten wie Festplattenpartitionen aufgebaut sind. Es genügt, die Startspur auf 0 und den Endspur auf eine Zahl zu setzen, die größer oder gleich der letzten Spur der Diskette ist ($FFFF_{16}$ erfüllt diesen Zweck ganz sicher). Die Kontroll-ISR für den Interrupt 13_{16} muß diesen Kunstgriff natürlich berücksichtigen. Durch die Gleichbehandlung läßt sich der Programmieraufwand stark verringern.

```
int im_i_table (char source[])
{ FILE *in;                             /* Liste Partitionsrechte  */
  char  drive[3];                       /* Laufwerk                */
  char  rights_str[17];                 /* Rechtestring            */
  int   i;
  BYTE  *n_ptr, far *f_ptr;              /* zur Datenuebertragung  */
  BYTE  p_drive;                         /* physikalisches Laufwerk*/
  BYTE  l_drive;                         /* logisches Laufwerk     */
  BYTE  l2_drive;                        /* temp. log. Laufwerk    */

  /* Diskettenlaufwerk-"Partitionen" anlegen                       */
  i_table.start_part[0] = 0;
  i_table.part[0].start = 0;
  i_table.part[0].end   = 0xFFFF;
  i_table.start_part[1] = 1;
  i_table.part[1].start = 0;
  i_table.part[1].end   = 0xFFFF;
```

Der nächste Teil ist stark mit `ReadPart` verwandt und arbeitet auf gleiche Weise. Vor jedem Aufruf von `read_part_data` wird der Verweis auf die erste Partition des gerade bearbeiteten physikalischen Laufwerks in `start_part` eingetragen.

```
  /* Teil 1: Partitionsdaten einlesen                              */
  p_drive = 0x80;                       /* Start mit 1. Festplatte */
  l_drive = 2;                          /* "A:", "B:" ueberspringen*/
  do
  { i_table.start_part[2 + (p_drive & 0x7F)] = l_drive;
  } while (read_part_data (&l_drive, p_drive++, 0, 0, 1) == 0);
```

Die Partitionsdaten werden diesmal nicht nur wie in `ReadPart` als Text ausgegeben, sondern in die Tabelle `i_table` geschrieben. `read_part_data` wurde so modifiziert, daß die Funktion für jedes logische Laufwerk die Start- und Endspur vermerkt. Das folgende Codefragment zeigt das Ende der Funktion unmittelbar nach der `switch`-Anweisung.

```
    /* normale Partition?                                          */
    if ((type == 0x01) || (type == 0x04) || (type == 0x06))
    { i_table.part[*ldrive].start =
        (GET_TRACK (mbr.part[i].start_combi) << 4) +
        mbr.part[i].start_head;
      i_table.part[*ldrive].end   =
        (GET_TRACK (mbr.part[i].end_combi) << 4) +
```

```
        mbr.part[i].end_head + 1;
      (*ldrive)++;                      /* inkr. log. Laufwerksnr. */
      if (*ldrive >= M_DRIVE)           /* Limit ueberschritten?   */
      { fprintf (stderr, "More than 10 logical drives -> abort\n");
        return (-1);
      };
    };
  };
  return (0);
};
```

Zurück zu `im_i_table`. Nach Ermittlung der Partitionsdaten wird die Liste der Partitionsrechte `p_rights.lst` zeilenweise gelesen und analog zu `compile_f_rights` konvertiert.

```
  /* Teil 2: Partitionsrechte lesen und konvertieren              */
  if ((in = fopen (source, "r")) == NULL)
  { fprintf (stderr, "Couldn't open partition rights file\n");
    return (-1);
  };

  while (fscanf (in, "%2s %16s",
    drive, rights_str) == 2)
  { /* Plausibilitaetspruefung                                    */
    l2_drive = tolower (drive[0]) - 'a';
    if ((0 > l2_drive) || (l2_drive >= l_drive))
    {
      fprintf (stderr, "Illegal drive specification: \"%c:\"\n",
        'A' + l2_drive);
    }
    else
    { /* konvertiere Rechte in Eintrag                            */
      i_table.part[l2_drive].rights =
        tokenise (rights_str, "           rwfbm");
    };
  };
  fclose (in);
```

Ein Aufruf von `intercom` fragt die FAR-Adresse der Partitionstabelle innerhalb von `AVWatchP` ab. Falls der Watcher korrekt antwortet, kopiert `im_i_table` die Tabelle in das residente Programm.

```
  /* bestimmte Adresse der Partitionstabelle in AVWatch*          */
  msg.prg_id = 'p';
  msg.pwd    = PWD;
  f_ptr = (BYTE far *)&msg;
  if ((intercom (IM_I_TABLE, (void far *)&f_ptr) & 0xFF) !=
      IM_I_TABLE)
  { fprintf (stderr, "AVWatch doesn't respond.\n");
    return (-2);
  };
  /* uebertrage Daten zu AVWatch*                                 */
  n_ptr = (BYTE *)&i_table;
  for (i = 0; i < sizeof (struct T_I_TABLE); i++)
  { *(f_ptr++) = *(n_ptr++);
  };
  return (0);
};
```

Wenn man sich die Größe von `im_i_table` betrachtet, konnte durch die Auslagerung aus `AVWatchP` eine Menge Arbeitsspeicher gespart werden.

im_deinst. Sehr einfach verläuft die Deinstallierung eines residenten Programms. `im_deinst` fragt vom Benutzer den Kennbuchstaben für den Watcher und das Paßwort ab und übermittelt per `intercom` den Befehl, sich aus dem Speicher zu entfernen. Die Bestätigung kommt vom Watcher selbst. Zwei Fehlersituationen sind möglich: Das Paßwort ist falsch oder der Watcher war gar nicht installiert.

```
int im_deinst (void)
{ WORD result;
  char c;

  printf ("Input program-ID (character) > ");
  c = getch ();
  msg.prg_id = c;
  printf ("\nInput password > ");
  scanf  ("%X", &msg.pwd);
  result = intercom (IM_DEINST, (void far *)&msg_ptr);
  if (result == ((c << 8) | IM_INVAL_PWD))
  { fprintf (stderr, "Invalid password. ");
  };
  if (result != ((c << 8) | IM_DEINST))
  { fprintf (stderr, "Deinstallation failed (i.e. TSR not found).");
  };
  msg.pwd = 0l;                          /* Passwort ruecksetzen    */
  printf ("\n");
  return (result == ((c << 8)| IM_DEINST));
};
```

mod_seal. Die schon behandelte Funktion `chk_seal` überprüft die Integrität des Programms, in dem sie sich befindet. Dazu bildet sie die aktuelle Prüfsumme und vergleicht diese mit der gespeicherten Referenzsumme am Ende der Programmdatei. `chk_seal` erzeugt aus Gründen der Platzersparnis diese Daten nicht selbst, sondern ist auf Hilfe von außen angewiesen. Dafür ist `AVConfig` mit der Funktion `mod_seal` zuständig.

Am Anfang der Funktion wird versucht, das Siegel der vom Benutzer spezifizierten Programmdatei in die Struktur `seal` einzulesen.

```
int mod_seal (void)
{ struct T_SEAL seal;
  FILE *in;
  char f_name[65];

  printf ("Attach / remove program seal\n\n"
          "Program name > ");
  scanf  ("%s", f_name);

  /* oeffnen fuer Kontrolle                                        */
  if ((in = fopen (f_name, "rb")) == NULL)
  { fprintf (stderr, "Couldn't open file\n");
    return (-1);
  };
  /* positioniere auf Programmsiegel                               */
  fseek (in, -(long)sizeof (struct T_SEAL), SEEK_END);
  if (fread (&seal, sizeof (struct T_SEAL), 1, in) != 1)
  { fprintf (stderr, "Couldn't read file seal\n");
```

```
    fclose (in);
    return (-2);
  };
  fclose (in);
```

Anhand der im Siegel enthaltenen Identifikationsnummer 2342_{16} kann das Programm entscheiden, ob das Siegel vorhanden ist oder nicht. Je nach Ergebnis bietet `mod_seal` dem Benutzer an, ein Siegel zu installieren (`attach_seal`), oder ein bestehendes Siegel zu entfernen (`remove_seal`).

```
  if (seal.id != 0x2342)
  { printf ("This program carries no seal.\n"
            "Attach seal (y)es/NO > ");
    if (tolower (getch ()) == 'y')
    { return (attach_seal (f_name));
    };
    return (0);
  }
  else
  { printf ("This program allready carries a seal\n"
            "Remove seal (y)es/NO > ");
    if (tolower (getch ()) == 'y')
    { return (remove_seal (f_name));
    };
    return (0);
  };
};
```

Hinweis. Durch das Hinzufügen oder Entfernen des Programmsiegels verändert sich die Prüfsumme der gesamten Programmdatei. `chk_seal` stört das nicht, weil es das angehängte Siegel nicht mit in die Berechnung der Signatur einbezieht. Programme wie `ChkState` und `AVWatchI` bemerken aber sehr wohl, daß sich etwas verändert hat und die Integrität der Datei verletzt wurde. Außerdem geht bei der Neuübersetzung eines sich selbst überprüfenden Programms das Siegel natürlich verloren, selbst wenn nichts am Code geändert wurde. Nach der Kompilierung ist mit `AVConfig` der Schutz wieder aufzufrischen.

attach_seal. Das Anbringen des Siegels ist vergleichsweise einfach, weil MS-DOS wie die meisten Betriebssysteme das Erweitern von Dateien am Ende unterstützt. `attach_seal` füllt `seal` mit der Identifikationsnummer und der Prüfsumme und fügt das so generierte Siegel an die Programmdatei an.

```
int attach_seal (char f_name[])
{ BYTE buf[BUF_SIZE];                  /* CRC-Puffer              */
  struct T_SEAL seal;                  /* Programmsiegel          */
  FILE *out;                           /* Programmdatei           */

  /* berechne Programmsiegel                                      */
  seal.id  = 0x2342;
  seal.sig = CRC_START;
  make_crc_table (POLYNOMIAL);
  if (sig_crc (buf, BUF_SIZE, f_name, &seal.sig))
  { fprintf (stderr, "Couldn't compute seal\n");
    return (-3);
  };

  /* haenge Programmsiegel an Datei an                            */
```

```
  if ((out = fopen (f_name, "ab")) == NULL)
  { fprintf (stderr, "Couldn't open file to attach seal\n");
    return (-4);
  };
  if (fwrite (&seal, sizeof (struct T_SEAL), 1, out) != 1)
  { fprintf (stderr, "Couldn't attach seal\n");
    return (-5);
  };
  fclose (out);
  return (0);
};
```

remove_seal. Etwas komplizierter ist das Entfernen der Prüfdaten, weil keine Funktion zum Verkürzen einer Datei zur Verfügung steht. Deshalb muß `remove_seal` das ganze Programm in eine temporäre Datei umkopieren, wobei die letzten Bytes mit dem Siegel weggelassen werden.

```
int remove_seal (char f_name[])
{ FILE *in, *out;                       /* Programm-/Hilfsdatei    */
  BYTE buf[BUF_SIZE];                   /* CRC-Puffer              */
  long to_go;                           /* zu schreibende Bytes    */
  WORD bytes;                           /* Anzahl gelesene Bytes   */

  /* lege temporaere Datei an                                       */
  if ((out = fopen ("avconfig.tmp" , "wb")) == NULL)
  { fprintf (stderr, "Can't create temporary file\n");
    return (-6);
  };

  /* kopiere Programm in temp. Datei, schneide Siegel ab            */
  if ((in = fopen (f_name , "rb")) == NULL)
  { fprintf (stderr, "Cant open program file\n");
    fclose (out);
    unlink ("avconfig.tmp");
    return (-7);
  };
  fseek (in, 0, SEEK_END);
  to_go = ftell (in) - sizeof (struct T_SEAL);
  rewind (in);
  while ((bytes = fread (buf, 1, min (to_go, BUF_SIZE), in)) != 0)
  { if (fwrite (buf, 1, bytes, out) != bytes)
    { fprintf (stderr, "Write error (copying file)\n");
    };
    to_go -= bytes;
  };

  fclose (in);
  fclose (out);
```

Die ursprüngliche Programmdatei wird gelöscht und die Hilfsdatei per Umbenennung zur Programmdatei gemacht:

```
  if (unlink (f_name))
  { fprintf (stderr, "Couldn't delete temporary file\n");
  };
  if (rename ("avconfig.tmp", f_name))
  { fprintf (stderr,
      "Couldn't rename temporary file in program file\n");
    return (-8);
  };
```

```
  return (0);
};
```

Kapitel 5

Diskussion

> The only truly secure system is one that is powered off, cast in a block of concrete, and sealed in a lead-lined vault with armed guards — and even then I have my doubts.
>
> Gene Spafford zitiert in *VIRUS-L Digest Vol.3 No.121*

Mit dem Erreichen des 5. Kapitels liegt die Theorie und Praxis der Abwehr von Softwareanomalien zunächst einmal hinter uns. Wir haben aus theoretischen Überlegungen heraus ein Paket von Funktionen und Programmen entwickelt, das wir nun rückblickend kritisch unter die Lupe nehmen wollen. Inwiefern wurden die selbstgesteckten Ziele erreicht, welcher Schutz wird geboten und wo liegen potentielle Schwächen?

Der Ausblick versucht aufzuzeigen, wie die zukünftige Situation sowohl auf dem Virensektor als auch auf dem Gebiet der Abwehrprogramme aussehen könnte. Welche Entwicklungen zeichnen sich ab, wie können die eigenen Programme sinnvoll erweitert werden? Nicht zuletzt werden auch einige skurrile, erschreckende, interessante und humorvolle Aspekte von Computerviren vorgestellt.

5.1 Einschränkungen

5.1.1 Abwehr

Betrachten wir der Reihe nach die Schutzprogramme, ihre Wirkungsweise und die Art der Schutzmaßnahmen. Da wäre zunächst die Gruppe der nicht-residenten Programme, zu der die Prüfprogramme `ChkSys`, `Seal`, `chk_seal` und `ChkState` sowie die neuen externen Kommandos `AVCopy` und `AVRename` zählen. Dazu kommen noch die Hilfsprogramme `ReadMCB` und `ReadPart`, die zwar keinen Schutzcharakter haben, aber dennoch Auskunft über den Zustand des Systems geben. Ein aufmerksamer Benutzer kann unerwünschte Manipulationen so evtl. “mit bloßem Auge” erkennen (zu Virensuche

mit konventionellen Werkzeugen s.a. [36]). Die zweite Gruppe stellt die der residenten Programme `AVWatchG, -I, -P` und `-F` dar, die auf unterschiedliche Weise und auf verschiedenen Ebenen des Betriebssystems arbeiten.

ChkSys. Das Programm `ChkSys` sucht nach verdächtigen Dateien, ohne Informationen über konkrete Viren zu benötigen. Der Benutzer wird auf Dateien und Verzeichnisse aufmerksam gemacht, die in einer "schwarzen Liste" stehen oder durch ungewöhnliche Attribute auffallen.

Seal und chk_seal. Mit `Seal` können Prüfsummen über Dateien, Datenträger und Urladeinformationen erzeugt werden. Der Anwender kann durch den manuellen Vergleich von Soll- und Ist-Wert Manipulationen z.B. auf dem Transportweg erkennen. `chk_seal` dient zur automatischen Selbstüberprüfung von Programmen beim Start.

ChkState. Der Dateibestand wird von dem Programm `ChkState` überwacht, das gleichzeitig die Referenzliste für `AVWatchI` liefert. Festgehalten werden die Struktur des Dateisystems, der Bestand an Dateien, Dateiattribute und — stellvertretend über eine Signatur — der Dateiinhalt. Regelmäßig angewendet liefert `ChkState` Informationen über Veränderungen im Dateisystem, die dem Benutzer als Report zur Beurteilung vorgelegt werden.

Schwachpunkt: Stealth-Viren. Aktive Viren, die Interruptaufrufe abfangen, um den korrekten Zustand einer Datei oder des System vorzuspiegeln, lassen sich von Programmen, die Dateien untersuchen, nicht entdecken. In diese Kategorie fallen alle Scanner und Checker, wobei `ChkState` zur letzten Gruppe gehört. Das einzige, worauf sich ein Schutzprogramm bei aktiviertem Stealth-Virus noch verlassen kann, sind Zugriffe auf den Arbeitsspeicher. Aus diesem Grund überprüfen gute Scanprogramme vor dem Suchlauf, ob sich ein Virus im Speicher installiert hat. Alternativ dazu genügt es, vor dem Start des Prüfprogramms den Rechner auszuschalten und von einer sauberen Diskette urzuladen — damit ist der erste Schwachpunkt behoben.

AVCopy. Das Kopierprogramm `AVCopy` verhindert, daß verseuchte Programme überhaupt auf den Rechner gelangen. Dabei wird nur zwischen Text und anderen Daten unterschieden, d.h. `AVCopy` ist nicht auf bestimmte Softwareanomalien zurechtgeschnitten. Unabhängig von der Erweiterung des Dateinamens kann z.B. die Übertragung von Daten unbestimmten Inhalts (= nicht Text) verhindert werden. Andere Programme mit ähnlicher Intention wie `VCopy` von McAfee Associates überprüfen wie ein Scanner die zu kopierenden Dateien auf konkrete Viren, aber eben mit dem Nachteil, daß nur bekannte Viren ausgesondert werden können.

Schwachpunkt: Kodierung. Das Konzept von `AVCopy` und allen anderen Scanprogrammen versagt, falls die zu überprüfende Datei in codierter Form vorliegt. Immerhin würde `AVCopy` auch z.B. gepackte Dateien abweisen, weil diese nicht die Merkmale eines Textes aufweisen. Mit etwas Aufwand ist es aber möglich, beliebige Dateien in Textform zu bringen.

Eine Methode wird im Anhang B vorgestellt: die Umcodierung mit dem Programm `uuencode`, das den Wertebereich der einzelnen Bytes einer Datei transformiert. Aus drei Bytes mit Werten zwischen 0 und 255 werden vier Bytes mit Werten von 32 bis 95, was ganz normalen Textzeichen entspricht. Sinn der Aktion ist es, Programm-

dateien problemlos und ohne Datenverfälschung über elektronische Netze übertragen zu können. `uuencode` und das zugehörige Dekodierprogramm `uudecode` sind als Public Domain-Quelltexte in vielen Programmiersprachen für jedermann verfügbar.

Schwachpunkt: Quellcode. Ein Virus oder ein beliebiges anderes Programm (z.B. `uudecode` oder ein Kopierprogramm) kann auch als Quellcode auf den Rechner gebracht werden. `AVCopy` kann zwar das Vorhandensein von Text erkennen, nicht aber dessen Bedeutung. Eingeschränkt wird diese Methode dadurch, daß sich auf dem System ein Compiler oder Interpreter befinden muß, der den kopierten Quelltext übersetzt oder ausführt.

Die ultimative Tarnmethode hat sich Peter Wayner (`wayner@cs.cornell.edu`) ausgedacht ([**?**] 11.71). Sein Programm, das er auf Anfrage versendet, verschlüsselt Daten in Texten, die z.B. wie eine Rede von Neil Kinnock oder eine Baseballreportage aussehen (sic). Der Zieltext mit der verschlüsselten Information ist nach bestimmten Regeln entsprechend typischer Merkmale eines längeren Quelltextes aufgebaut. Das Ergebnis ist etwas konfus und ohne Sinn, aber da man das auch von manchen Politikern oder Sportreportern gewohnt ist, trotzdem unauffällig.

AVRename. Das Umbenennungsprogramm erfüllt unterstützende Aufgaben. Zum einen wird verhindert, daß Dateien ihren Typ wechseln und z.B. ein getarntes Programm ausführbar oder ein Programm getarnt wird. Zum anderen kann ein Programm nicht umbenannt oder in ein anderes Verzeichnis transportiert werden, um z.B. die Überwachung durch `AVWatchF` zu unterlaufen.

AVWatchG. Der Schutz der Systemuhr gewährleistet die Korrektheit von Zeitmarken in Logdateien. Manipulationen könnten es unmöglich machen, den Verursacher einer unzulässigen Operation zu ermitteln.

Schwachpunkt: Direktzugriffe. Die eingebaute Hardware-Systemuhr kann auch über Portadressen programmiert werden, was übrigens allgemein für den Uhrenchip und die Daten im CMOS-RAM gilt. Diese Zugriffe sind im Unterschied zu den BIOS-Aufrufen nicht kontrollierbar. Mögliche Abhilfe bietet die regelmäßige Prüfsummenbildung über die Konfigurationsdaten und eine Plausibilitätsprüfung für die Uhrzeit.

AVWatchI. Dieser Watcher überprüft *vor* dem Start eines Programms dessen Integrität anhand von Daten aus einer Referenzliste. Ein z.B. durch ein Virus infiziertes und damit verändertes Programm kommt erst gar nicht zur Ausführung; das Virus wird nicht aktiviert. Dies ist die primäre Schutzfunktion. Selbst wenn ein verseuchtes Programm versehentlich in die Referenzliste aufgenommen wird, werden sekundäre Infektionen und Manipulationen erkannt. Ein Virus kann `AVWatchI` auf zwei Arten angreifen: Täuschung des Watchers oder Manipulation der Referenzdaten. Die Verfälschung der Kommunikationswege durch Stealth-Viren haben wir bereits bei `ChkState` angesprochen.

Schwachpunkt: Rechtemanipulation. Alle Prüfprogramme, die anhand des Inhalts bestimmter Dateien Entscheidungen über die Rechtmäßigkeit einer Operation treffen, sind von deren Integrität abhängig. Rechte- und Referenzlisten dürfen daher

für Fremdprogramme nicht zugänglich sein. Unter MS-DOS sind folgende Maßnahmen denkbar:

- Verstecken der Datei durch Setzen des `hidden`-Attributs und/oder Kontrolle der Kernelfunktionen zum Suchen und Öffnen von Dateien.
- Blockierung aller Schreibzugriffe durch Setzen des `readonly`-Flags und/oder Kontrolle der Aufrufe zum Ändern der Dateiattribute und Öffnen von Dateien.

AVWatchP. Schreib- und Leseschutz auf Partitionebene wird von `AVWatchP` realisiert. Die Überwachung kann auf die Urladeinformation (MBR und PBR) beschränkt werden, um normale Zugriffe zuzulassen, Urladeviren aber einen Riegel vorzuschieben.

Schwachstelle: Direkter Zugriff. Besonders für BSIs, die Manipulationen meist nur sektorweise durchführen, ist die direkte Programmierung des Festplattencontrollers eine günstige Alternative zu BIOS-Aufrufen. Die Manipulierung von Ports und Speicheradressen läßt sich nämlich nur hardwaremäßig kontrollieren, wofür in vielen Computern MMUs (**M**emory **M**anagement **U**nits) zuständig sind. Der Atari ST und der Commodore Amiga (68000-kompatible CPUs) sind beide mit solchen Speichermanagern ausgerüstet, die unzulässige Zugriffe erkennen und Speicherbereiche auf logische Adressen abbilden. Auch die INTEL-CPUs im Protected Mode verfügen über Speicherschutzmechanismen, die sich allerdings unter MS-DOS eines permanenten Ruhestands erfreuen.

AVWatchF. Zugriffe auf Dateien werden von `AVWatchF` kontrolliert. Wie die meisten Watcher wirkt `AVWatchF` "post infectionem" und soll nach der Aktivierung eines Virus Schlimmeres, d.h. weitere Infektionen und andere Manipulationen, verhüten. Die Methode des direkten Zugriffs auf die Hardware kommt hier weniger in Frage, weil das betrachtete Objekt nicht ein Sektor, sondern eine rel. komplex organisierte Datei ist.

Schwächen allgemein. Ein so ungesichertes Betriebssystem wie MS-DOS bietet Softwareanomalien viele Möglichkeiten, in interne Abläufe einzugreifen oder an der Systemsoftware vorbeizuarbeiten. Das ist kein Fehler, der Schutzprogrammen vorgeworfen werden kann, aber der durch andere Betriebssysteme und Hardwaremaßnahmen behoben werden sollte.

Schutz- und Selbstschutzfunktionen können durch Programme des Benutzers ausgeschaltet oder umgangen werden, wenn er diese als Quelltext auf die Festplatte bringt, dort kompiliert und schließlich benutzt. Diese Methode wurde bereits am Beispiel `uuencode` erläutert. Das mithin sicherste System ist daher eines, das keine Software zur Programmerstellung enthält und nur ungefährliche Daten mit der Umgebung austauscht. Die Allgemeinheit der Interpretation macht eine Präzisierung notwendig: "gefährlich" sind Programme, die sicherheitsrelevante Operationen ausführen können; die Interpretation "gefährlicher" Daten veranlaßt diese Operationen.

Grad der Zielerreichung. Hier gilt es, zwei verschiedene Einsatzarten der Schutzsoftware zu betrachten. Der private PC-Anwender wird dafür sorgen, daß die Antivirusprogramme, die er installiert hat, auch korrekt arbeiten. Für ihn würde die kontrollierte Isolation eher eine Behinderung seiner Arbeit bedeuten, der Schutz der Programmintegrität wäre völlig ausreichend.

In einem Rechenzentrum oder Betrieb hingegen ist mit Aktionen der Benutzer gegen die Schutzsoftware zu rechnen. Diese kann auf normalen MS-DOS-Rechnern keine völlige Sicherheit bieten, aber helfen, die Grenze zwischen "versehentlich" und "absichtlich" deutlich zu ziehen. Mit der Installation der Programme und Durchführung der vorgeschlagenen Maßnahmen ist es nicht mehr möglich, "versehentlich" Programme von Diskette auszuführen oder auf Festplatte zu kopieren. Gegen einen Benutzer, der einen Virus trotz Schutzmaßnahmen auf den Rechner bringt, kann bei Entdeckung ganz anders vorgegangen werden als gegen einen, der an einem gewöhnlichen System arbeitet.

Zudem gilt für die meisten Anwender, daß Schutzmaßnahmen gegen Schusseligkeit und Leichtsinn völlig ausreichend sind. Dies wird mit der kontrollierten Isolation erreicht. Andere Benutzer benötigen eine kleine Gedächtnisstütze, was Regeln im Umgang mit fremden Programmen und dem firmeneigenen Computer angeht. Eine Warnung des Kontrollsystems kann dies bieten. Unter Berücksichtigung der Tatsache, daß kein Konzept auf Softwarebasis gegen einen gezielten Angriff oder das Umgehen von Schutzmaßnahmen sicher schützen kann, wurden die gesetzten Anforderungen erreicht.

5.1.2 Selbstschutz

Das Wächterprogramm muß gegen Angriffe geschützt werden, falls der Einsatz in "feindlicher" Umgebung geplant ist. Hierbei zeigen sich zwei Hauptangriffspunkte: Der Start des Wächterprogramms und der Erhalt der Funktionsfähigkeit.

Sicherer Start. Wenn der Rechner eingeschaltet wird, muß das Wächterprogramm vor allen anderen (Nicht-System-) Programmen aktiviert werden, was am einfachsten über die `autoexec.bat`-Datei erfolgt. Deren Ausführung kann aber durch Ctrl Break unterbrochen oder durch Urladen von Diskette völlig umgangen werden. Der Abbruch der Stapeldatei läßt sich durch Modifikation von `command.com` verhindern. Gegen den Start von Diskette sind softwaregestützte Maßnahmen machtlos.

Funktionsfähigkeit. Es existieren Software-Werkzeuge, mit denen TSR-Programme deaktiviert und/oder entfernt werden können. Diese sollten sich normalerweise nicht auf dem System befinden. Theoretisch verhindern `AVWatchF`, `AVCopy` und `AVRename`, daß derartige Programme gestartet werden können. Dieser Schutz ist aber, wie oben gezeigt, nicht unfehlbar.

Deaktivierung. Eine Deaktivierung erfolgt dadurch, daß die in das TSR-Programm zeigenden Interruptvektoren auf andere ISRs umgesetzt werden. Die einzige (schwache) Maßnahme dagegen besteht darin, sich in möglichst viele Interrupts einzuklinken und bei einem Aufruf alle anderen Vektoren zu kontrollieren und notfalls wieder auf das eigene Programm zu setzen. Neben Programmen wie `sidekick` benutzen manche Viren diese Technik [9]. Gleichbedeutend mit Deaktivierung ist die Direktzugriffsmethode, die ebenfalls die Kontroll-ISRs des Watchers umgeht.

Fazit: Solange Benutzer den Ablauf des Schutzprogramms in irgendeiner Weise beeinflussen können, ist das System nicht sicher. Manipulationen am Schutzprogramm und den zugehörigen Dateien müssen unmöglich sein. Dieses Problem kann mit einem

Fileserver gelöst werden, der für die Benutzer räumlich nicht zugänglich ist. Auf diesem können zentrale Sicherheitsmaßnahmen implementiert werden. Alle angeschlossenen APCs verfügen über ein spezielles ROM-BIOS, welches das Betriebssystem ausschließlich vom Server lädt. Programmstarts von Diskette und das Einspielen von Programmen sind nicht zulässig (kontrollierte Isolation). Falls Diskettenlaufwerke nicht unbedingt erforderlich sind, sollten sie ganz weggelassen werden (vollständige Isolation).

5.2 Ausblick

5.2.1 Zukünftige Entwicklung

Viren tauchen in immer größerer Anzahl und Vielfalt der Typen auf. 1990 vollzog sich ein Generationswechsel von mehr oder weniger simplen Viren zu sog. Sophisticated (engl.: raffiniert) Viruses. Diese sind in der Lage, sich vor dem Benutzer zu verbergen und Schutzprogramme zu umgehen, indem sie Stealth (Tarn-) Techniken verwenden, sich selbst verschlüsseln, mutieren und am Interrupt-Konzept von MS-DOS vorbei arbeiten. Bei "Whale" kann keine noch so kurze einfache Bytesequenz das Virus identifizieren [55]. Ein anderes Virus ist in der Lage festzustellen, ob ein Programm im Speicher nach ihm sucht. Ist das der Fall, stürzt der Rechner ab; der Verdacht fällt evtl. auf das Suchprogramm. Über Viren, die sich selbst verbessern und ihr "Wissen" erweitern können, wird schon diskutiert ([38] 3.149, 156, 157, 158). Ist angesichts dieser etwas düsteren Entwicklung `AVSystem` nicht schon bald überholt?

Pluspunkt: Sättigungsangriff. Das entwickelte Konzept ist von den Fähigkeiten zukünftiger Viren oder anderer Softwareanomalien nicht abhängig, da es nicht auf bestimmte Eigenschaften derselben zurechtgeschnitten ist. Scanner veralten zwangsläufig mit der Entwicklung neuer Viren. Es ist durchaus denkbar, daß ein überaktiver Zeitgenosse einen cleveren Virusgenerator entwickelt, mit dessen Hilfe er Unmengen verschiedenartiger Viren in Umlauf bringt. Falls die Exemplare stark genug differieren und damit unterschiedliche Suchstrings und Desinfektionsmethoden notwendig werden, ist das Ende der Scanner in Sicht. Vielleicht wird dieses Szenario sowieso bald Wirklichkeit, wenn die Anzahl der Viren weiter so wächst wie 1991.

Schutzkonzepte, die zulassen, daß ein Virus überhaupt aktiv wird, müssen durch Watcher unter großem Aufwand Barrieren errichten, die von Virenprogrammierern (von Viren selbst?) analysiert und überwunden werden können. Konzepte wie die kontrollierte Isolation und die Überprüfung der Programmintegrität vor dem Start werden tätig, bevor das Virus die Möglichkeit zum Handeln bekommt.

Pluspunkt: Schutz der Systemintegrität. Die einzige erfolgversprechende Methode scheint die des Integritätsschutzes zu sein, der auf unterster Systemebene ansetzen und Bestandteil des Betriebssystems sein muß. Durch hardwaremäßige Verschlüsselung und Integritätsprüfung spielt die zusätzliche Verarbeitungszeit kaum eine Rolle. Man könnte sogar Verfahren zur Datenkompression auf Sektorebene integrieren[1], die die Kapazität eines Datenträgers erhöhen und die Ladezeiten verkürzen.

[1] von DR-DOS 6.0 schon softwaremäßig realisiert.

Die Schwierigkeit bei der Integritätsüberwachung besteht darin, daß, um vernünftig mit dem System arbeiten zu können, Dateien auch erstellt, geändert und gelöscht werden müssen. Doch welche Instanz ist dazu berechtigt und welche nicht? Cohen's Experimente haben ja deutlich gezeigt, wie sich Viren erfolgreich der Rechte ihrer Wirtsprogramme und damit der Anwender bedienen. Ebenso bringt uns die Authentifizierung des Anwenders nicht weiter, denn selbst eine durch Paßwort und Chipkarte legitimierte Person kann Sabotage betreiben. Daher müssen Meldungen des Systems über manipulierte Dateien unveränderbar protokolliert werden. Für den Kontrollzugriff auf die Datei und für die Kommunikation mit dem Anwender sind sichere Kanäle erforderlich, denn sonst könnten Softwareanomalien (bes. Stealth-Viren) den Benutzer oder die Schutzsoftware täuschen.

Allgemein läßt sich ein Trend zu verbesserter Zugriffskontrolle und Schutz der Datenintegrität auch, oder gerade, auf PCs feststellen. Betriebssysteme wie das DOS 6.0 für PCs von Digital Research realisieren einen Paßwortschutz und die Überwachung von kritischen Operationen. Mit dem Aufkommen von speziellen Hardwarebausteinen zur Ver- und Entschlüsselung von Daten stellt umfassender Integritätsschutz kein Hindernis mehr dar, auch, was den Aufwand an Rechenleistung betrifft.

5.2.2 Sinnvolle Erweiterungen

Das mit `AV-System` realisierte Schutzkonzept läßt sich relativ leicht auch auf andere DOS-Kommandos, Kernel-Funktionen und Interrupts ausweiten. Um weitere Befehle zu überwachen, müssen interne Kommandos externisiert und externe Kommandos ersetzt werden. Für zusätzliche Interrupt-Kontrollen stehen `Std_TSR` und `Std_INTC` zur Verfügung.

An den konventionellen Programmen gibt es an sich nicht viel zu verbessern, es sei denn, der Komfort für den Anwender soll mit einer ansprechenden Benutzeroberfläche und einem Hilfesystem erhöht werden. Bei den residenten Programmen gäbe es noch eine Menge zu tun. Als Erstes wäre die vollständige Umsetzung in Assembler zu nennen, um den Platzbedarf und den Verbrauch an Rechenzeit auf ein Minimum zu reduzieren. Der Autor würde allerdings nicht versuchen, die Funktionen der Watcher noch stark auszuweiten. Ein Virus oder trojanisches Pferd kann bereits beim ersten Aufruf das System mit einem Schlag verwüsten, wenn es an MS-DOS vorbei arbeitet und z.B. direkt den Festplattencontroller programmiert. Die beste Maßnahmenkombination ist wohl — Stichwort: Programmintegrität — die von `AVWatchI` und `ChkState`. Vorausgesetzt, daß nur geprüfte Programme ins System übernommen werden, ist der Rechner tatsächlich virensicher. Mit Hilfe der genannten Programme besteht zusätzlich die gute Chance, daß dies so bleibt.

5.2.3 "Computerviren, das Universum und der ganze Rest"

Computerviren sind eine ernste Bedrohung der Rechnersicherheit, allemal ein interessantes Thema und zugleich ein Gebiet eigenartiger Motive und Überlegungen. Der

folgende Text gibt Antworten auf Fragen, die in den vorangegangenen Kapiteln noch nicht behandelt wurden.

Wie alt sind Computerviren? Eine strittige Frage, deren Beantwortung nicht zuletzt davon abhängt, wie man den Begriff "Computervirus" definiert. Cohen war mit Sicherheit nicht der Erste, der einen Virus programmierte, wohl aber der Pionier der wissenschaftlichen Erforschung dieses Phänomens. Der Ausdruck "Computervirus" stammt von Cohen's Betreuer Len Adleman, der sich u.a. einen Namen mit der Entwicklung des RSA- und MD4RSA-Verfahrens gemacht hat.

Von 1983 stammt der Vorschlag eines Quelltextvirus (Source Code Virus) unter UNIX, obwohl dieser Name damals noch nicht verwendet wurde. Ken Thompson, einer der Entwickler von UNIX, entwickelte theoretisch ein Verfahren, wie in das für das Einloggen zuständige Programm `login` eine Falltür eingebaut werden kann. Die Überlegung verläuft in drei Schritten:

1. In `login` wird direkt die Falltür eingebaut; der Quelltext soll nicht verändert werden. Nachteil: Bei der nächsten Kompilierung liegt das Programm wieder in der Originalversion vor.
2. Der "C"-Compiler `cc` wird so verändert, daß er bei der Kompilierung von `login` den Trojaner einbaut. Nachteil: Auch der Compiler kann aus dem Original-Quelltext kompiliert werden.
3. Der Compiler wird so modifiziert, daß er bei der Kompilierung seines Quelltextes eine Kopie der drei Manipulationsfunktionen in den neuen Compiler aufnimmt.

Mit der letzten Veränderung erfüllt der Manipulationsteil des Compilers die Definition eines Virus: Er verändert ein Programm (das gerade erzeugte) so, daß es eine Kopie seiner selbst enthält. Demnach lag bereits 1983 das Konzept für ein raffiniertes Computervirus aus berufenem Munde vor.
(Quelle: [38] 3.130, 3.132)

Können Computerviren zufällig entstehen? Anfang 1991 haben bulgarische Virenprogrammierer ein funktionsfähiges Virus mit einer Länge von 42 (zweiundvierzig) Bytes entwickelt[2]. Soll das Virus durch Bitfehler entstehen, ist die Wahrscheinlichkeit einer zufälligen Erzeugung bei Gleichverteilung der Bytewerte $\frac{1}{256^{42}} \approx 10^{-101}$. Ob das wenig ist oder nicht, hängt von der Häufigkeit der Erzeugung der 42 Bytes ab. Jedes ca. 10^{101}te mal entsteht statistisch gesehen einmal das Virus. Weil die Erfolgsaussichten eher gering sind, ordnet Cohen dieses Problem in dieselbe Kategorie wie die Frage "Wie wahrscheinlich ist es, daß ein Affe einen Virus schreibt?" ein.

Da Bereiche eines existierenden Programms verfälscht werden können, die schon einen Teil des Virus darstellen, oder weil nicht unbedingt genau die Reihenfolge der Bytes (= Befehle) eingehalten werden muß, erhöht sich die Wahrscheinlichkeit um ein unbekanntes Maß. Die große Anzahl von Computern und Datenträgern auf der Welt, auf denen potentiell Daten verfälscht werden, bedingt eine hohe Anzahl von Zufallsversuchen pro Tag. Alles in allem hängt eine Aussage über die Wahrscheinlichkeit

[2] Im Ostblock scheint auch RAM knapp zu sein.

einer zufälligen Entstehung von zu vielen unbekannten Faktoren ab, ist aber eher als sehr unwahrscheinlich anzusehen.
(Quelle: Cohen, eigene Überlegungen)

Gibt es eine Evolution unter Computerviren? Evolution ist ein ständig wiederholter Prozeß von Mutation und Auslese. Negativ veränderte Individuen (→ nicht funktionsfähige Computerviren) sterben aus, positiv veränderte überleben und verdrängen die schwächeren. Eine positive Mutation auf dem Gebiet der Computerviren wäre z.B. schon die Veränderung eines Bytes, die das Virus nicht verkrüppelt, aber für aktuell gebräuchliche Scanner nicht erkennbar macht. Für weitergehende, echte Modifikationen der Funktion eines Virus gelten ähnliche Überlegungen wie im vorherigen Abschnitt.

Eine ganz andere Sache sind mutierende Viren, die sich selbst verändern, um ihre Entdeckung durch Scanner zu behindern. Das "Whale"-Virus spickt die Dekodiersequenz für den Hauptcode mit unnützen Befehlen, ändert, soweit das möglich ist, die Reihenfolge der Kommandos und benutzt verschiedene Schlüssel. Auf diese Weise wird erreicht, daß sich zwei "Whale"-Exemplare u.U. in keinem Byte gleichen [55].

Eignen sich Computerviren für das Militär? Praktisch alles findet Anwendung auf militärischem Gebiet, fast nichts könnte nicht auch für Zwecke der Kriegsführung mißbraucht werden. Bestimmte Anforderungen an eine Waffe existieren nicht, sie sollte nur möglichst dem Gegner und nicht der eigenen Truppe schaden. Warum dann nicht Computerviren verwenden, die sich doch so gut für die Vernichtung von Daten und Rechenkapazität eignen? Diese Frage stellte das amerikanische Verteidigungsministerium (DoD) sich und jedem Patrioten, der meint, zur Entwicklung eines "Kampf-Virus" beitragen zu können. In der mit US$50000 dotierten Ausschreibung (Kürzel "CV ECM"[3]) wird vor allen Dingen nach einer Methode gesucht, um das Virus drahtlos in feindliche Systeme zu implantieren. Dem Gewinner stehen US$500000 zur Entwicklung zur Verfügung. Dazu die Stimme eines Diskussionsteilnehmers auf `VIRUS-L`:

> I don't think they could be used as a battlefield weapon ("Quick Lieutenant, fire that Brain virus, that's the only thing that'll stop 'em now!")
>
> Charles Cafrelli (`iaqr100@indyvax`) [38] 3.092)

Im Golfkrieg Anfang 1991 kam es tatsächlich zur Behinderung von militärischen Operationen durch Computerviren. Diese wurden allerdings von amerikanischen Soldaten eingeschleppt, die sich vom Gefechtsdienst mit Computerspielen (Ballerei auf dem Bildschirm?) erholen wollten.
(Quelle: [90], [38] 3.090ff)

Sabotage-Viren. Während des Golfkrieges kam auf der Virusliste eine von Prof. Klaus Brunnstein initiierte Diskussion in Gang, ob exportierte Waffen Mechanismen, insbesondere Viren enthalten könnten, um den Einsatz gegen das exportierende

[3] **C**omputer **V**irus **E**lectronical **C**ounter **M**easures

Land oder befreundete Nationen zu verhindern. Das Beispiel der Versenkung des britischen Zerstörers "Sheffield" im Falklandkrieg gegen Argentinien durch eine französische "Exocet"-Rakete zeigt, daß dies in der Vergangenheit zumindest nicht der Fall gewesen ist. Mit diesen Betrachtungen wollen wir die vom Autor ungeliebte militärische Seite hinter uns lassen.
(Quelle: [38] 4.010)

Wo kommen sie her? Viren kommen von überall her. Offensichtlich gibt es selbst auf Island und Neuseeland genügend Einwohner und zu wenig trennendes Wasser, denn von dort her stammen die Viren "Islandic" bzw. "Stoned". Einer der ersten Viren überhaupt, "Brain", kommt aus Lahore in Pakistan (Nahost); "Denzuk" schmückt die Kennung eines indonesischen Radioamateurs (Fernost)[4]. Die UdSSR, Ungarn und besonders Bulgarien (Osteuropa) sind in letzter Zeit zu Virenquellen erster Ordnung geworden. Die Produktion von Viren und die Verbreitung über BBS ist dort weder strafbar noch wird sie verfolgt. Es ist aber zu erwarten, daß mit westlichen Wirtschaftshilfen für Länder des Ost-"blocks" die Forderung nach entsprechenden Gesetzen seitens der Geldgeber verbunden sein wird. Die momentane Situation könnte sich dadurch langfristig ändern, wenn auch momentan (1991) aus diesen Staaten eine wahre Flutwelle von Viren in die westliche Welt überschwappt und den Antivirus-Forschern das Leben schwer macht. Damit kein Vorwurf von Einseitigkeit aufkommt: Auch in Deutschland und Großbritannien gibt es mehrere Virus-BBS, die das Ausland mit zu Recht finsteren Blicken im Auge behält.
(Quelle: [38] 4.035)

Warum sind sie hier? Betrachtet man die Wirkungsweise diverser Viren, zeigt sich, daß nichts zu ausgefallen, schrill oder merkwürdig sein kann, als daß es nicht schon programmiert worden wäre (Tab. 5.1). Die Ziele reichen von der gezielten Schädigung des Benutzers ("Armageddon") über politische Parolen ("Stoned", "Saddam") bis hin zu reinen Jux ("Ambulance", "Loa Duong").
(Quelle: [38])

Viren als Einkommensquelle. Zu unterscheiden sind zwei Parteien: diejenigen, die Viren schreiben und die, die Programme zur ihrer Abwehr entwickeln. Die Antivirus-Front hat es, kraft der Legalität ihres Tuns, leichter. Viren sind ein Sicherheitsrisiko, das den Anwender zur Anschaffung von Gegenmitteln, d.h. Sicherheitssoftware und Antivirusprogrammen, veranlaßt. McAfee Associates z.B. ist ein Konzern, der sicherlich vom Verkauf von Antivirusprogrammen leben könnte. Andere Autoren vertreiben ihre Produkte als Shareware oder gar als Freeware und verdienen damit nichts oder relativ wenig. Auf der Gegenseite bieten manche BBS gegen Bezahlung Quelltexte von Viren an. In einigen Zeitschriften und Büchern wurden und werden Viren zum Abtippen veröffentlicht, wohl um damit manchem einen Anreiz zum Kauf zu bieten.

Wie immer gehen die Meinungen darüber auseinander, ob sich das Aussetzen von neuen Computerviren für Hersteller von Antivirusprodukten lohnen würde. Ein Gegenargument ist, daß sich neue Viren kaum oder nur sehr langsam verbreiten und sich deswegen niemand sofort eine neues Abwehrprogramm zulegt. Auf lange Sicht

[4] Kennung: "YC1ERP", Bandung, Indonesien [38] 3.134
[5] Auch bei Fernseh- und Kinofilmen möglich, aber verboten.

Virus	Funktion
Ambulance	Ein Krankenwagen fährt mit Sirenengeheul über die Bildschirn
Armageddon	Versucht regelmäßig, die griechische Zeitansage anzurufen (Selbstwählmodem und Aufenthalt in Griechenland vorausgesetzt, kann es teuer werden)
Cascade	Es wird Herbst: Die Buchstaben fallen herab und sammeln sich am Boden des Bildschirms
Eight Tunes	Juke-Box mit 8 Songs, Auswahl zufällig
Holland Girl	Enthält die Aufforderung, an "Sylvia" in Holland zu schreiben
Loa Duong	Verbreitet fremde Kultur, indem es den gleichnamigen laotischen Grabgesang spielt
Sublimal	Die Nachricht "Love, remember?" wird in gewissen Abständen so kurz eingeblendet, daß nur das Unterbewußtsein sie wahrnimmt[5]
Sunday	Dieser Virus meint es gut mit dem Anwender und fordert ihn, falls es Sonntag ist, auf, sich zu schonen
Stoned	Der Bob Marley der Viren: "Legalize it!" auf elektronischem Wege
Saddam	Propaganda gegen den bekannten irakischen Staatsmann

Tabelle 5.1: Viren-Spezialitäten

hingegen könnte sich diese Strategie auszahlen. Für die betreffende Firma bestünde allerdings das große Risiko, im Falle der Aufdeckung ihrer Machenschaften einigen Prozessen gegenüberzustehen.

Ein Pro-Argument ist, daß ohne neue Viren ein Update-Service überflüssig wird. Viele Anwender sind vielleicht auch nervös genug, um auf Anzeigen wie "Neues Virus entdeckt — bestellen Sie Ihr Schutzprogramm *jetzt*!" zu reagieren. Alles in allem ist es völlig unnötig, selbst für neue Viren zu sorgen und dabei unwägbare Risiken auf sich zu nehmen. Hunderte unverantwortlicher Programmierer arbeiten weltweit Tag und Nacht freiwillig und ohne Bezahlung daran, daß der Virenstrom nicht abreißt. Warum sie ihre offensichtlichen Fähigkeiten nicht für Geld vermarkten, ist nicht oder nur schwer nachzuvollziehen.

Virenbekämpfung mit künstlicher Intelligenz. Wenn die natürliche Findigkeit nicht mehr ausreicht, wird die künstliche Intelligenz (AI = **A**rtificial **I**ntelligence) zu Rate gezogen. Cohen hat gezeigt, daß es nicht möglich ist, den Zweck eines Programms

im voraus zu erkennen. Das bedeutet aber nicht, daß dies mit einer bestimmten Trefferquote durchaus funktionieren kann. Es ist auf algorithmischem Wege nur sehr schwierig, ein so vage definierbares Ding wie einen Computervirus zu erkennen. Aus der automatischen Bilderkennung sind ähnliche Probleme bekannt: Was ist ein entgegenkommendes Auto, ein Raketensilo, eine Lärche?

Aktuelle Forschungsvorhaben beschäftigen sich mit neuronalen Netzen, die sich besonders zur Lösung unscharf definierter Probleme eignen. Ein solches Netz wird nicht programmiert, es bekommt einen Sachverhalt beigebracht. In vielen Lernschritten (Teach In) werden Objekte vorgeführt, die gewünschte Antwort vom Ausgang her ins Netz gefüttert und damit das Wissen verankert (Back Propagation). Bei einer ausreichenden Anzahl von künstlichen Nervenzellen lassen sich erstaunlich gute Ergebnisse erzielen.

Der Haken liegt bei der "ausreichenden Anzahl", der Komplexität einer Nervenzelle und besonders bei der Vielzahl von Verbindungen zwischen den einzelnen Zellen, die eine praktische Umsetzung stark erschweren. Zukünftige Technologien wie optische Computer kommen den Anforderungen neuronaler Netzwerke sehr entgegen, befinden sich aber noch in einem frühen Stadium der Entwicklung. Ein entsprechend ausgebildeter Mensch kann anhand des Codes immer bestimmen, ob ein Programm das tut, was es seiner Beschreibung nach tun soll. Prinzipiell kann auch ein sehr großes neuronales

Netz das leisten, was sein Vorbild, das (menschliche) Gehirn, kann. Bis dahin ist es — zum Glück? — noch ein weiter Weg.

Vergleichsweise konventionell aufgebaut sind Expertensysteme (XPS = E**xp**ert **S**ystem), in denen das Wissen eines oder mehrerer Experten in Form von logischen Regeln gespeichert ist. Stark vereinfacht ausgedrückt: "if (Programm ist angeblich PacMan) and (Programm enthält Formatierbefehle) then (Programm ist Trojaner)". Expertensysteme werden erfolgreich z.B. bei der Wartung von Flugzeugtriebwerken eingesetzt (MRCA Tornado bei MBB). Der Techniker gibt Symptome und Meßwerte ein, das Expertensystem führt ihn dabei, fragt je nach Antwort nach weiteren Daten und gibt letztendlich eine Prognose über die Fehlerursache ab. Zwei prinzipielle Anwendungen von XPS sind denkbar: Beratung des Anwenders bei Virusproblemen und die automatische Erkennung gefährlicher Programme.

Im ersten Fall würden (hoffentlich) kompetente Antivirusforscher ein System erstellen, das den Anwender beim Schutz und der Bekämpfung von Computerviren rein theoretisch unterstützt. Vom System erteilte Ratschläge werden stellvertretend durch den Benutzer ausgeführt. Fragen und Antworten können recht abstrakt gehalten sein, da sie durch einem Menschen interpretiert werden. Dies ist die wahrscheinlichste und vermutlich sinnvollste Anwendung eines XPS bei der Abwehr von Softwareanomalien.

Anders ist die Situation bei der selbständigen Erkennung von Softwareanomalien. Daten über das zu untersuchende Programm müssen von einer Informationsbeschaffungskomponente des Expertensystems gewonnen werden. Sonst wäre der Anwender gezwungen, sich selbst Expertenwissen anzueignen und das Programm vor der Sitzung genau zu untersuchen. Das Verfahren mit der automatischen Informationsgewinnung versagt allerdings, wenn der Programmtext verschlüsselt und die Dekodierroutine gut versteckt ist. Einziger Hinweis auf das eventuelle Vorliegen einer Softwareanomalie ist die Anwesenheit von verschlüsseltem Code und der Dekodierfunktion.

Über die automatische Analyse hinaus kann der Anwender vom XPS über das Programm befragt werden. Auf diese Weise ergibt sich das oben angeführte, sehr stark vereinfachte Beispiel. Die Analyse des Programms ergibt Hinweise auf verwendete Befehle, bestimmte Codesequenzen und andere Eigenschaften. Die Antworten des Programmbenutzers geben dem XPS Aufschluß über die hinter dem Programm stehende Intention, den Verwendungszweck. Aus beiden Faktengebieten zusammen ist für das XPS evtl. eine Gefährdung erkennbar.

Sind Viren auszurotten? Die Fragen "Woher kommen Computerviren? Warum sind sie hier?" wurden bereits geklärt. Wichtiger erscheint die Antwort auf die Frage "Wohin gehen sie?". Ist die einmal geöffnete Büchse der Pandora wieder verschließbar und läßt sich das freigelassene Übel wieder einfangen? Auf der Virusliste entspann sich eine Diskussion, die das praktische Verschwinden der Pocken mit Computerviren in Verbindung brachte. Dabei stellten sich einige wesentliche Unterschiede heraus:

- Die Pocken wurden durch eine globale Aktion der Weltgesundheitsorganisation (WHO) besiegt. Eine ähnliche Behörde auf dem Computersektor existiert zur Zeit nicht.

- Der (einzige) Pockenerreger kann durch Impfung wirksam abgewehrt werden. Dagegen existiert eine Vielzahl von Computerviren mit einer großen Neuentstehungsrate, gegen die es a priori keine Gegenmaßnahmen gibt (s. Cohen's Theorien).
- Pockenerreger können außerhalb des Körpers nicht überleben. Auf Disketten und anderen Speichermedien können Computerviren beliebig lange Zeiten außerhalb (sauberer) Rechner überdauern.

Immerhin: Obwohl Seuchen wie die Cholera keineswegs ausgerottet sind, verschwand in modernen Staaten durch die dort übliche Frischwasserversorgung der Hauptgrund für die Verbreitung. Einzelne Erkrankte werden isoliert und ärztlich behandelt. In Ballungsgebieten, in denen z.B. wegen eines Erdbebens die Versorgung mit sauberem Wasser zusammengebrochen ist, entwickeln sich häufig Cholera-Epidemien. Analog zur Seuchenbekämpfung könnte ein Integritätsschutz für saubere Programme sorgen, obwohl Verseuchung weiterhin möglich ist. Bei der nächsten Benutzung des Programms wird diese sofort erkannt, und Gegenmaßnahmen können eingeleitet werden.

Anhang A

Software

A.1 Die Begleitdiskette

Auf der Begleitdiskette befinden sich die Quelltexte sämtlicher Programme und Funktionen, die in diesem Buch besprochen werden. Bestellen können Sie die Diskette mit der dem Buch beigefügten Bestellkarte. Die im Text eingefügten Quelltexte stellen eine z.T. stark gekürzte und u.U. veränderte Version der Originalquelltexte dar. Auf der Diskette wurden insbesondere Leerzeilen zur Strukturierung eingesetzt, die Kommentierung ist ausführlicher und erfolgt in englischer Sprache.

Inhalt. Im Verzeichnis `SOURCE` befinden sich die Quelltexte der Programme, in `AVSYS` und `MSDOS_S` die Quelltexte zu den gleichnamigen Systembibliotheken. Für jede "C"-Datei existiert eine korrespondierende Definitionsdatei (`*.h`) in `INCLUDE` und eine Projektdatei (`*.prj`) in `PRJ`.

Die Projektdateien sind nur dann von Bedeutung, wenn einzelne Programme mit der integrierten Entwicklungsumgebung `tc.exe` kompiliert werden sollen. In diesem Fall ist die Projektdatei, eine Art `makefile`, explizit anzugeben (Menü `Project`, Unterpunkt `Project name`). Diese legt fest, aus welchen Modulen außer dem Startmodul und den Standardbibliotheken das Programm aufgebaut ist. Dazu gehören die Programmodule des Anwenders und zusätzliche Module und Bibliotheken wie `avsys.lib` und `msdos_s.lib`. Beispielsweise enthält die Projektdatei für `AVWatchI`, `avwatchi.prj`, folgende Angaben:

```
avwatchi.obj
avsys.lib
msdos_s.lib
```

Für die Kompilierung mit der Entwicklungsumgebung sind einige Compiler- und Linker-Optionen (Menü `Options`, Untermenü `Compiler` bzw. `Linker`) einzustellen:

- Compiler: `Code generation`

  - `Alignment: Byte` (bei `Alignment: Word` würden die Strukturen in `msdos.h` falsch behandelt)
  - `Merge duplicate strings: Off` und
  - `Test stack overflow: Off` (führt sonst bei TSR-Programmen zu Problemen)
  - Rest auf `On`
- `Optimization`
  - `Optimize: Size`
  - Rest auf `Off` (andernfalls wird der Code sehr undurchsichtig)
- `Source`
  - `Identifier length: 32` (Standard)
  - Rest auf `Off` (wir benötigen z.B. dir non-ANSI-Bezeichner `near` und `far`)
- `Errors`
  - alles auf `On` (bei der TSR-Programmierung ist jeder Hinweis hilfreich)
- Linker
  - `Stack Warning: On` (warnt bei TSR-Programmen (Speichermodell `TINY`), daß diese noch in `com`-Dateien umzuwandeln sind)
  - `Warn duplicate symbols: On` (s.a. nächster Punkt)
  - Rest auf `Off` (`Case-sensitive link` muß `Off` sein, weil `tasm` Symbole in Großbuchstaben produziert (z.B. in den Bibliotheken), `tc` aber normalerweise alle Symbole in Kleinbuchstaben erwartet.)

Für die automatische Generierung der Bibliotheksdateien und Programme mit `make` sind im jeweiligen Verzeichnis `makefile`s verfügbar. Für die Leser, die nicht über einen Compiler oder Assembler verfügen, befinden sich außerdem die fertig übersetzten und gebrauchsfertigen Programme und Bibliotheken in den Verzeichnissen `PACKAGE` bzw. `LIB`. Diejenigen, die selbst Schutzprogramme erstellen möchten, können unter Angabe der Quelle auf die Bibliotheken zurückgreifen.

Für die "C"-Programme stehen drei `#include`-Dateien zur Verfügung:

- `avsys.h` enthält Definitionen und Deklarationen für die Bibliothek `avsys.lib`. In dieser sind außer den Betriebssystemaufrufen alle Funktionen enthalten, die das Paket AV-System benötigt.
- `msdos_s.h` ist die Definitionsdatei für die Bibliothek `msdos_s.lib`, die "C"-Funktionen für alle benötigten MS-DOS-Aufrufe enthält. Für Leser, die nicht über Turbo-C verfügen, stellt `msdos_s.lib` funktional identische Module bereit.
- `msdos.h` enthält Definitionen und Strukturen, welche die Arbeit mit MS-DOS betreffen. Vor allen Dingen handelt es sich dabei um "C"-Strukturen, die den Aufbau MS-DOS-interner Datenstrukturen nachbilden und den Zugriff darauf vereinfachen.

Dazu kommen ausgewählte Dokumente in `DOCUMENT`, die aus Rechnernetzen stammen und z.T. schwierig oder gar nicht mehr zu beschaffen sind. Leser ohne Netzzugriff

erhalten so die Möglichkeit, einige interessante Texte zum Thema Softwareanomalien zu studieren.

Ablauf der Programmerstellung (über make):

1. Arbeitskopie der Originaldiskette anfertigen und für alle weiteren Operationen verwenden.
2. `PATH` so setzen, daß `tcc.exe`, `tasm.exe`, `tlink.exe`, `tlib.exe` und `make.exe` erreichbar sind.
3. Erzeugen `avsys.lib` in `LIB`: Wechseln in Verzeichnis `AVSYS`, Aufruf "`make`".
4. Erzeugen `msdos_s.lib` in `LIB`: Wechseln in Verzeichnis `MSDOS_S`, Aufruf "`make`".
5. Erzeugen der Programme in `PACKAGE`: Wechseln in Verzeichnis `SOURCE`, Aufruf "`make`".

Ablauf der Programmerstellung (integrierte Entwicklungsumgebung):

1. – 4. s.o.
5. `tc.exe` aufrufen, Namen der Projektdatei eingeben.
6. Optionen für `Compiler` und `Linker` wie beschrieben setzen.
7. Kompilieren.

Inhaltsübersicht Begleitdiskette:

- `SOURCE`: Programme und Beispielprogramme
  - Quelltexte aller Programme und Beispielprogramme
  - `makefile` für die automatische Generierung
- `AVSYS`: AV-Systembibliothek, Speichermodell `SMALL`
  - Quelltexte `avsys.lib`
  - `makefile` für die automatische Generierung
  - Dateiliste `file.lst` für `makefile`
- `MSDOS_S`: MS-DOS-Bibliothek, Speichermodell `SMALL`
  - Quelltexte `msdos_s.lib` (Ersatzfunktionen für Turbo-C Routinen, zusätzliche MS-DOS-Funktionen)
  - `makefile` für die automatische Generierung
  - Dateiliste `file.lst` für `makefile`
- `INCLUDE`: `include`-Dateien
  - <Programmname>`.h`
  - `avsys.h`
  - `msdos_s.h`
  - `msdos.h` (Datenstrukturen unter MS-DOS)
- `PRJ`: Projektdateien für die Kompilierung der Programme mit Turbo-C (integrierte Entwicklungsumgebung)

  - <Programmname>.prj
- LIB: gebrauchsfertige Bibliotheksdateien
  - avsys.lib
  - msdos_s.lib
- PACKAGE: gebrauchsfertige Programme
- DOCUMENT: Dokumente aus Rechnernetzen zum Thema Softwareanomalien, Betriebssysteminterna von MS-DOS
  - invntory.doc (Kurzbeschreibung des Inhalts der einzelnen Texte)
  - Kurzanleitungen zu den wichtigsten Servertypen (engl.)

A.2 Software zur Programmerstellung

Turbo-C (C-Compiler). Alle "C"-Quellcodes (*.c) werden mit dem Turbo-C-Compiler übersetzt, wobei entweder die integrierte Entwicklungsumgebung tc.exe oder die Kommandozeilenversion tcc.exe zum Einsatz kommt. Im ersten Fall kann auf separate Kompilierungs- und Linkläufe verzichtet werden. Die Kommandozeilenversion wird beim Einsatz von make benötigt.

Bei den TSR-Programmen ist die resultierende exe-Datei mit dem externen Kommando exe2bin in eine bin-Datei umzuwandeln, die vom Aufbau her mit dem com-Typ identisch ist. Nach der Umbenennung in eine com-Datei ist das Programm betriebsbereit. Bei Benutzung von make werden diese Schritte automatisch erledigt. Alternativ dazu kann bei exe2bin auch gleich die Zieldatei angegeben werden. Beispiel:

```
exe2bin avwatchi avwatchi.com
```

Turbo-Assembler (Assembler). Der Turbo-Assembler tasm übersetzt die in Maschinensprache geschriebenen Module (Endung asm) in obj-Dateien. Da nur Bibliotheksmodule in Assembler implementiert wurden, kommt der Anwender mit tasm nicht in Berührung. Die Übersetzung und Zusammenstellung der Bibliotheksmodule erfolgt durch den Aufruf von make.

Turbo-Lib (Librarian). Die Module der Bibliotheken avsys.lib (größtenteils in "C") und msdos_s.lib (in Assembler) werden mit tlib zusammengefügt.

Turbo-Link (Linker). Der Linker tlink verbindet einzelne Module (u.a. aus Bibliotheken) zu einem lauffähigen Programm. Die Entwicklungsumgebung tc ruft den Linker automatisch auf, so daß die Aktivierung von tlink praktisch unsichtbar bleibt. Falls die Programme und Bibliotheken per make generiert werden, kommen die Kommandozeilenversion tcc und der Linker separat zum Einsatz.

Turbo-Debug (Debugger). Ein Programm ist entweder mit Fehlern behaftet oder trivial. Deshalb ist ein Debugger besonders in solchen Fällen von Nöten, wo die Inspektion der Quelltexte allein nicht mehr weiterhilft. Bei der Suche nach Ursachen für Fehler im Linkvorgang erwies sich das Programm tdump als besonders nützlich, mit dem sich interne Informationen in obj- und exe-Dateien auflisten lassen. Zu Fehlern, die ohne Debugger nur schwer zu entdecken sind, ein paar Beispiele:

- Variablenübergabe (falsche Typen → unterschiedliche Länge und Interpretation; bes. bei NEAR- und FAR-Zeigern),
- Laufzeitfehler (Speicherverbrauch, Stacküberlauf, Reentrancy-Probleme),
- Fehler im Linkvorgang (falsche Deklaration von Segmenten).

Alle angeführten Programme sind von Borland. `Flushot`, `ViruScan`, `F-Prot` und viele andere Antivirusprogramme können über `trickle@ds0rus1i`, Verzeichnis `<MSDOS.-TROJAN-PRO>`, bezogen werden (s.a. B.4 “TRICKLE-Server”).

A.3 MSDOS_S.LIB

A.3.1 Kernel-Interrupts

Allgemeine Hinweise. Zu jeder Funktion der Bibliothek `MSDOS_S.LIB` existiert eine kombinierte Funktionsbeschreibung für Assembler (erste Spalte) und "C" (zweite Spalte). Die dritte Spalte enthält eine Beschreibung oder den Wert der Parameter und geht auf Besonderheiten der "C"-Version ein.

Des weiteren sind folgende Dinge zu beachten:

- Alle Module sind für das Speichermodell `SMALL` ausgelegt; der Einsatz ist auch im Modell `TINY` möglich.
- Die "C"-Funktionen setzen automatisch die NEAR-Adressen übergebener Parameter in FAR-Adressen um, falls der Assembler-Aufruf das verlangt (Speichermodell `SMALL`!). Die Übergabe erfolgt immer wie im Prototyp spezifiziert.
- In "C"-Programme eingebundene Assembler-Module müssen den Inhalt der Register `DS`, `SI` und `DI` erhalten, weil der Compiler dies erwartet.
- Die Funktion `absread` hinterläßt das Flagregister auf dem Stack. Die "C"-Routine behebt diesen Fehler automatisch.
- `CF` steht für Carry-Flag, `ZF` für Zero-Flag.
- `return` steht für den Wert, den die "C"-Funktion zurückgibt.

Funktion "Get Current Disk"
Interrupt 21_{16}, Funktion 19_{16}
`int xgetdisk (void)`

Aufrufparameter:

`AH`		19_{16}

Seiteneffekte:
keine

Rückgabeparameter:

`AL`	`return`	aktuelles Laufwerk (0: `A:...`)

Tabelle A.1: Get Current Disk (INT 21_{16}, Funktion 19_{16})

Funktion "Get Drive Data":
Interrupt 21_{16}, Funktion $1C_{16}$
`int drivedata(int drive, BYTE *spc, BYTE far **media, WORD *bps, WORD *cpd)`

Aufrufparameter:

`AH`		$1C_{16}$
`DL`	`drive`	Laufwerksnummer (0: aktuell; 1: `A:`...)

Seiteneffekte:
`*spc`, `*media`, `*bps` und `*cpd` werden gesetzt

Rückgabeparameter:

`AL`	return	FF_{16}: Fehler
`AL`	`*spc`	Sektoren pro Cluster
`DS:BX`	`*media`	FAR-Zeiger auf Media Descriptor Byte
`CX`	`*bps`	Bytes pro Sektor
`DX`	`*cpd`	Cluster pro Drive (Laufwerk)

Tabelle A.2: Get Drive Data (INT 21_{16}, Funktion $1C_{16}$)

Funktion "Set Interrupt Vector":
Interrupt 21_{16}, Funktion 25_{16}
`void xsetvect (int interruptno, void interrupt (*isr) ())`

Aufrufparameter:

`AH`		25_{16}
`AL`	`interruptno`	Interruptnummer
`DS:DX`	`isr`	FAR-Adresse der ISR

Seiteneffekte:
Der spezifizierte Interruptvektor wird auf die neue ISR gesetzt

Rückgabeparameter:
keine

Tabelle A.3: Set Interrupt Vector (INT 21_{16}, Funktion 25_{16})

Funktion "Parse Filename":
Interrupt 21_{16}, Funktion 29_{16}
`char *xparsfnm (const char *cmdline, struct T_FCB *fcb, int option)`

Aufrufparameter:

`AH`		29_{16}
`AL`	`option`	Optionen (auf 0 setzen)
`DS:SI`	`cmdline`	FAR-Adresse Dateiname
`ES:DI`	fcb	FAR-Adresse FCB

Seiteneffekte:
Der Dateiname wird analysiert und in den FCB übertragen

Rückgabeparameter:

`AL`		00_{16}: keine Jokerzeichen; 01_{16}: Jokerzeichen; FF_{16}: fehlerhafte Laufwerksangabe
`DS:SI`	`return`	Zeigt auf erstes Zeichen im Dateinamen nach Ende Analysevorgang (Analyse wird durch Trennzeichen wie Leerzeichen, Semikolon etc. beendet)

Tabelle A.4: Parse Filename (INT 21_{16}, Funktion 29_{16})

Funktion "Get Date":
`void xgetdate (struct T_DATE *date)`
Interrupt 21_{16}, Funktion $2A_{16}$

Aufrufparameter:

`AH`	$2A_{16}$

Seiteneffekte:

`*date`	aktuelles Systemdatum

Rückgabeparameter:

`CX`	`date -> year`	Jahr
`DH`	`date -> month`	Monat
`DL`	`date -> day`	Tag
`AL`		Wochentag (0: Sonntag...)

Tabelle A.5: Get Date (INT 21_{16}, Funktion $2A_{16}$)

Funktion "Get Time":
`void xgettime (struct T_TIME *time)`
Interrupt 21_{16}, Funktion $2C_{16}$

Aufrufparameter:

`AH`		$2C_{16}$

Seiteneffekte:

`*time`		aktuelle Systemzeit

Rückgabeparameter:

`CH`	`time -> hour`	Stunde
`CL`	`time -> minute`	Minuten
`DH`	`time -> second`	Sekunden
`DL`	`time -> centi`	Zehntelsekunden

Tabelle A.6: Get Time (INT 21_{16}, Funktion $2C_{16}$)

Funktion "Terminate and Stay Resident":
Interrupt 21_{16}, Funktion 31_{16}
`void xkeep (unsigned char status, unsigned size)`

Aufrufparameter:

`AH`		31_{16}
`AL`	`status`	Rückgabewert (des Programms)
`DX`	`size`	Größe des zu reservierenden Speichers (in Paragraphs)

Seiteneffekte:
Das Programm wird beendet, der Speicher jedoch nicht freigegeben

Rückgabeparameter:
keine (keine Rückkehr!)

Tabelle A.7: Terminate and Stay Resident (INT 21_{16}, Funktion 31_{16})

Funktion "Get Interrupt Vector":
Interrupt 21_{16}, Funktion 35_{16}
`void interrupt (*xgetvect (int intr_num)) ()`

Aufrufparameter:

`AH`		35_{16}
`AL`	`intr_num`	Interruptnummer

Seiteneffekte:
keine

Rückgabeparameter:

`ES:BX`	`return`	FAR-Adresse der ISR

Tabelle A.8: Get Interrupt Vector (INT 21_{16}, Funktion 35_{16})

Funktion "Set Current Directory":
Interrupt 21_{16}, Funktion $3B_{16}$
`int xchdir (const char *path)`

Aufrufparameter:

`AH`		$3B_{16}$
`DS:DX`	`path`	FAR-Adresse Pfadname

Seiteneffekte:
Das aktuelle Verzeichnis wird festgelegt

Rückgabeparameter:

`CF`		0: kein Fehler; 1: Fehlercode in `AX`
`AX`	`return`	0: kein Fehler; n: Fehlercode

Tabelle A.9: Set Current Directory (INT 21_{16}, Funktion $3B_{16}$)

Funktion "Get Device Information":
Interrupt 21_{16}, Funktion 44_{16}, Subfunktion 00_{16}
`int xisatty (int handle)`

Aufrufparameter:

`AH`		44_{16}
`AL`		00_{16}
`BX`	`handle`	Handle

Seiteneffekte:
keine

Rückgabeparameter:

`CF`		0: Geräteinformation in `DX`; 1: Fehlercode in `AX`
`DX`	`return`	Bit 7: {0: Datei; 1: Gerät}
`AX`	`return`	Fehlercode

Tabelle A.10: Get Device Information (INT 21_{16}, Funktion 44_{16}, Subfunktion 00_{16})

Funktion "Get Current Directory":
Interrupt 21_{16}, Funktion 47_{16}
`int xgetcurdir (int drive, char *directory)`

Aufrufparameter:

`AH`		47_{16}
`DL`	`drive`	Laufwerksnummer (0: aktuell; 1: `A:`...)
`DS:SI`	`directory`	FAR-Adresse 64-Byte-Puffer

Seiteneffekte:
Das aktuelle Verzeichnis wird in den Puffer übertragen

Rückgabeparameter:

`CF`		0: kein Fehler; 1: Fehlercode in `AX`
`AX`	`return`	0: kein Fehler; n: Fehlercode

Tabelle A.11: Get Current Directory (INT 21_{16}, Funktion 47_{16})

Funktion "Release Memory Block":
Interrupt 21_{16}, Funktion 49_{16}
`int xfreemem (unsigned segx)`

Aufrufparameter:

`AH`		49_{16}
`ES`	`segx`	Segmentadresse des freizugebenden Blocks

Seiteneffekte:
Der spezifizierte Speicherblock wird freigegeben

Rückgabeparameter:

`CF`		0: kein Fehler; 1: Fehlercode in `AX`
`AX`	`return`	0: kein Fehler; n: Fehlercode

Tabelle A.12: Release Memory Block (INT 21_{16}, Funktion 49_{16})

Funktion "Load and Execute":
Interrupt 21_{16}, Funktion $4B_{16}$

Aufrufparameter:

`AH`		$4B_{16}$
`AL`		00_{16}: laden & ausführen; 03_{16}: nur laden
`ES:BX`		FAR-Adresse Parameterblock
`DS:DX`		FAR-Adresse Dateiname

Seiteneffekte:
Das spezifizierte Programm wird geladen und ausgeführt

Rückgabeparameter:

`CF`		0: kein Fehler; 1: Fehlercode in `AX`
`AX`	return	0: kein Fehler; n: Fehlercode

Tabelle A.13: Load and Execute (INT 21_{16}, Funktion $4B_{16}$)

Funktion "Find First File" und "Find Next File":
Interrupt 21_{16}, Funktionen $4E_{16}$ und $4F_{16}$
`int xfindfirst (const char *filename, struct T_DTA *dta, int attrib)`
`int xfindnext (struct T_DTA *dta)`

Aufrufparameter:

`AH`		$4E_{16}$ bzw. $4F_{16}$
`CX`	`attrib`	Suchattribut (Find First File)
`DS:DX`	`filename`	FAR-Adresse Suchmaske (Find First File)

Seiteneffekte:
Das aktuelle DTA wird mit Dateiinformationen gefüllt; die "C"-Funktion macht `dta` zum aktuellen DTA

Rückgabeparameter:

`CF`		0: kein Fehler; 1: Fehlercode in `AX`
`AX`	`return`	0: kein Fehler; n: Fehlercode

Tabelle A.14: Find First/Next File (INT 21_{16}, Funktion $4E/4F_{16}$)

Funktion "Get Address of List of Lists":
Interrupt 21_{16}, Funktion 52_{16}
`struct T_LOL far *getlol (void)`

Aufrufparameter:

`AH`	52_{16}

Seiteneffekte:
`keine`

Rückgabeparameter:

`ES:BX`	`return`	FAR-Zeiger auf die List of Lists (Offset-Korrektur bei "C"-Funktion schon durchgeführt)

Tabelle A.15: Get Address of List of Lists (INT 21_{16}, Funktion 52_{16})

Funktion "Rename File":
int renfile (char old_file[], char new_file[])

Aufrufparameter:

AH		56_{16}
DS:DX	old_file	FAR-Adresse alter Dateiname
ES:DI	new_file	FAR-Adresse neuer Dateiname

Seiteneffekte:
Die spezifizierte Datei wird umbenannt

Rückgabeparameter:

CF		0: kein Fehler; 1: Fehlercode in AX
AX	return	0: kein Fehler; n: Fehlercode

Tabelle A.16: Rename File (INT 21_{16}, Funktion 56_{16})

Funktion "Get or Set File Date and Time":
Interrupt 21_{16}, Funktion 57_{16}
`int xgetftime (int handle, struct T_FTIME *ftime)`
`int xsetftime (int handle, struct T_FTIME *ftime)`

Aufrufparameter:

`AH`		57_{16}
`AL`		00_{16}: Get; 01_{16}: Set
`BX`	`handle`	Handle
`CX`	in `*ftime`	Zeit (Set)
`DX`	in `*ftime`	Datum (Set)

Seiteneffekte:
Set: Dateizeit und -datum werden gesetzt
Get: Dateizeit und -datum werden in `*ftime` übertragen

Rückgabeparameter:

`CF`		0: kein Fehler; 1: Fehlercode in `AX`
`CX`	in `*ftime`	Zeit (Get)
`DX`	in `*ftime`	Datum (Get)
`AX`	`return`	0: kein Fehler; n: Fehlercode

Tabelle A.17: Get or Set File Date and Time (INT 21_{16}, Funktion 57_{16})

Funktion "Get PSP Address":
Interrupt 21_{16}, Funktion 62_{16}
`unsigned xgetpsp (void)`

Aufrufparameter:

`AH`		62_{16}

Seiteneffekte:
keine

Rückgabeparameter:

`BX`	`return`	Segmentadresse PSP

Tabelle A.18: Get PSP Address (INT 21_{16}, Funktion 62_{16})

Funktion "Absolute Disk Read":
Interrupt 25_{16}
`int xabsread (int drive, int nsects, int lsect, void *buffer)`

Aufrufparameter:

`AL`	`drive`	logische Laufwerksnummer (0: `A:`...)
`CX`	`nsects`	Anzahl der zu lesenden Sektoren
`DX`	`lsect`	Startsektor
`DS:BX`	`buffer`	FAR-Adresse Puffer

Seiteneffekte:
Die angeforderten Sektoren werden in den Puffer geladen

Rückgabeparameter:

`CF`		0: kein Fehler; 1: Fehlercode in `AX`
`AX`	`return`	0: kein Fehler; n: Fehlercode

Tabelle A.19: Absolute Disk Read (INT 25_{16})

A.3.2 Video-Interrupt (BIOS)

Funktion "Set Cursor Position":
Interrupt 10_{16}, Funktion 02_{16}
void xgotoxy (int x, int y)

Aufrufparameter:

`AH`		02_{16}
`BH`	implizit 0	Videoseite
`DH`	`y`	Y (Reihe)
`DL`	`x`	X (Spalte)

Seiteneffekte:
Der Cursor auf Videoseite `BH` wird auf die Position (`x`, `y`) gesetzt

Rückgabeparameter:
keine

Tabelle A.20: Set Cursor Position (INT 10_{16}, Funktion 02_{16})

Funktion "Get Cursor Position":
Interrupt 10_{16}, Funktion 03_{16}
`xwherexy (int *x, int *y)`

Aufrufparameter:

`AH`		03_{16}
`BH`	implizit 0	Videoseite

Seiteneffekte:
`*x`, `*y` enthalten Cursorposition auf Videoseite `BH`

Rückgabeparameter:

`CH`		Startzeile Cursor
`CL`		Endzeile Cursor
`DH`	`*y`	Y (Zeile)
`DL`	`*x`	X (Spalte)

Tabelle A.21: Get Cursor Position (INT 10_{16}, Funktion 03_{16})

Funktion "Read Character and Attribute at Cursor":
Interrupt 10_{16}, Funktion 08_{16}
`WORD readchar (BYTE *char, BYTE *attr)`

Aufrufparameter:

`AH`		08_{16}
`BH`	implizit 0	Videoseite

Seiteneffekte:
`*char`, `*attr` enthalten Zeichen und Attribut an Cursorposition auf Videoseite `BH`

Rückgabeparameter:

`AH`	`*attr`	Attribut
`AL`	`*char`	Zeichen
`AX`	`return`	Kombinationsangabe `AH, AL`

Tabelle A.22: Read Character and Attribute at Cursor (INT 10_{16}, Funktion 08_{16})

Funktion "Write Character and Attribute at Cursor":
Interrupt 10_{16}, Funktion 09_{16}
`void writechar (BYTE char, BYTE attr)`

Aufrufparameter:

`AH`		09_{16}
`BH`	implizit 0	Videoseite
`AL`	`char`	Zeichen
`BL`	`attr`	Attribut
`CX`	implizit 1	Anzahl der Wiederholungen

Seiteneffekte:
Zeichenausgabe auf Bildschirm an Cursorposition auf Videoseite `BH`

Rückgabeparameter:
keine

Tabelle A.23: Write Character and Attribute at Cursor (INT 10_{16}, Funktion 09_{16})

A.3.3 Disk-Interrupt (BIOS)

Funktion "Disk-Interrupt":
Interrupt 13_{16}
int xbiosdisk (int cmd, int drive, int head, int track, int sector, int nsects, void *buffer)

Aufrufparameter:

AH		Funktionsnummer
AL	nsects	Anzahl der Sektoren
CH	track	Zylinder (Bits 0–7)
CL	sector	Sektor (Bits 6–7: Bits 8–9 von track
DH	head	Kopf
DL	drive	physikalisches Laufwerk (+ 80_{16}für Festplatten)
ES:BX	buffer	FAR-Adresse Puffer

Seiteneffekte:
Je nach Funktion

Rückgabeparameter:

CF		0: Anzahl übertragene Sektoren in AX; 1: Fehlercode in AX
AL	return	Anzahl übertragene Sektoren
AH	return	Fehlercode

Tabelle A.24: INT 13_{16}: "Disk-Interrupt"

A.3.4 Keyboard-Interrupt (BIOS)

Funktion "Keyboard-Interrupt":
Interrupt 16_{16}, Funktionen 00_{16}-02_{16}
`int xbioskey (int cmd)`

Aufrufparameter:

`AH`	`cmd`	Funktionsnummer (00_{16}: Zeichen lesen; 01_{16}: prüfen, ob Zeichen verfügbar; 02_{16}: Status Kontrolltasten lesen)

Seiteneffekte:
Funktion 00_{16}: Zeichen wird aus Tastaturpuffer entnommen

Rückgabeparameter:

`ZF`		Funktion 01_{16}: 0: Zeichen verfügbar; 1: kein Zeichen
`AH`		Scan-Code (Funktion 02_{16}: keine Bedeutung)
`AL`	`return`	ASCII-Code (Funktion 01_{16}: 0: kein Zeichen verfügbar; Funktion 02_{16}: Bitmuster für Kontrolltasten)

Tabelle A.25: INT 16_{16}: "Keyboard-Interrupt"

A.4 AVSys.LIB

Tabelle A.26 gibt eine Übersicht über die Funktionen der Bibliothek `AVSys.LIB`. Für den Fall, daß eine Funktion schon im Text besprochen wurde, gibt die Spalte "Abschnitt" neben der Seite auch den zugehörigen Unterpunkt an. Nicht besprochen werden die Routinen zur Umsetzung der DTA-Daten in Strings und zurück (s.a. Hinweis in 4.3.3 "ChkState").

A.4.1 Allgemein einsetzbare Funktionen

Funktionen für CRC-Prüfsummen. Das Paket `crc.c` enthält Funktionen zur Berechnung von Prüfsummen nach dem CRC-Verfahren (**C**yclic **R**edundancy **C**heck). Allerdings wollen wir uns hier nicht weiter mit der Theorie der Nachrichtenkodierung beschäftigen; wer möchte, kann genaueres in [**?**] nachlesen. Grundlage des CRC-Verfahrens ist ein sog. Generatorpolynom, mit dessen Hilfe die Prüfsumme berechnet wird. Gängige Generatorpolynome für Werte in Bytebreite sind in Tab. A.27 angegeben. In der Spalte "Code" stehen die Werte, wie sie die Funktion `make_crc_table` benötigt. Die höchstwertige Stelle des Codes repräsentiert den niederstwertigen Exponenten des Generatorpolynoms. Ein gesetztes Bit zeigt an, daß der korrespondierende Summand im Polynom vorkommt. Der höchste Exponent (16) wird, weil immer vorhanden, im Code nicht berücksichtigt und fällt weg.

Compute Signature (sig_crc, block_crc, make_crc_table). `sig_crc` basiert auf der Funktion `block_crc`, die eine Prüfsumme über den Inhalt des Puffers `buf` bildet. Mit `crc` kann ein Anfangswert angegeben werden, um z.B. mehrere Einzelberechnungen miteinander zu verknüpfen oder eine Art Paßwort einfließen zu lassen.

```
WORD block_crc (BYTE buf[], size_t n, WORD crc)
{ while (n--)
  { crc = (crc >> 8) ^ crc_table[(crc & 0xFF) ^ *(buf++)];
  };
  return (crc);
};
```

Vor dem ersten Aufruf von `block_crc` oder `sig_crc` und zum Wechsel des Generatorpolynoms ist die Funktion `make_crc_table` aufzurufen, die die globale Tabelle `crc_table` initialisiert. Durch die Vorberechnung kann `block_crc` die Prüfsumme anstatt bitweise gleich byteweise berechnen, was die Verarbeitung stark beschleunigt.

```
WORD crc_table[256];                    /* im CRC.OBJ global def.  */

void make_crc_table (WORD polynomial)
{ int value, bit;
  WORD result;

  for (value = 0; value <= 255; value++)
  { result = value;
    for (bit = 0; bit < 8; bit++)
    { result = (result >> 1) ^ ((result & 0x01) ? polynomial : 0);
    };
    crc_table[value] = result;
```

Funktion	Seite	Abschnitt
Allgemein einsetzbare Funktionen		
block_crc	281	
get_genotype	179	4.4.1 “AVCopy”
get_mcb_name	109	3.4.1 “Organisation des Arbeitsspeichers”
get_owner_mcb	110	3.4.1 “Organisation des Arbeitsspeichers”
get_phenotype	178	4.4.1 “AVCopy”
is_device	283	
make_crc_table	282	
message	284	
scan_dir	148	4.3.1 “Check System”
sig_crc	282	
tokenise	285	
Bearbeitung von Dateinamen		
add_path	286	
cmp_fname	287	
cmp_fspec	221	4.5.8 “AVWatchF”
comp_fspec	287	
fill_in	288	
norm_path	185	4.4.2 “AVRename”
norm_path2	183	4.4.2 “AVRename”
split_fspec	289	
Stringverarbeitung		
clean	290	
strcpyu	290	
stricmpu	290	
stristr	291	
Listenverwaltung		
add2list	292	
delete_list	292	
load_list	293	
select_p	293	
Zugriff auf die Kommandozeile		
arg_p	294	
get_arg	295	
Spezielle Funktionen für AV-System		
add2log	229	4.5.8 “AVWatchF”
check_seal	158	4.3.2 “Seal und Check Seal”
get_ref	298	
get_rights	300	
intercom	197	4.5.2 “Std_TSR”
put_ref	298	
read_t_rights	296	

Tabelle A.26: Funktionsübersicht AVSys.LIB

Polynom	Code	Bezeichnung
$X^{16}+X^{15}+X^2+1$	1010 0000 0000 0001 = $A001_{16}$	CRC-16
$X^{16}+X^{12}+X^5+1$	1000 0100 0000 1000 = 8404_{16}	CCITT

Tabelle A.27: Übliche Generatorpolynome

Funktion "Compute CRC-Checksum Over Block":
`WORD block_crc (BYTE buf[], size_t n, WORD crc)`

Aufrufparameter:

`buf`	Adresse Puffer
`n`	Anzahl Bytes
`crc`	Startwert

Seiteneffekte:
`keine`

Rückgabewert:
Prüfsumme über Puffer

Tabelle A.28: `block_crc`: Berechne CRC-Prüfsumme über Puffer

```
  };
};
```

`sig_crc` schließlich ist eine Anwendung von `block_crc` zur Berechnung der Prüfsumme über eine namentlich spezifizierte Datei. Das Polynom kann über `make_crc_table` vorgegeben werden. Der Startwert `crc` wird normalerweise mit 0 initialisiert; ein anderer Wert kann quasi als Paßwort dienen und erschwert dadurch z.B. Computerviren, die korrekte Prüfsumme einer veränderten Datei nachzubilden. In der `while`-Schleife wird von der Möglichkeit Gebrauch gemacht, die Berechnung der Prüfsumme über einen großen Block (Datei) in mehrere kleine Abschnitte zu zerlegen. Das ist auch notwendig, weil die "C"-Funktion `read` nicht mehr als 32767 Bytes mit einem Aufruf einlesen kann.

```
int sig_crc (BYTE buf[], size_t buf_size, char f_spec[], WORD *crc)
{ size_t bytes;                         /* Anzahl gelesene Bytes   */
  int handle;                           /* Dateihandle             */

  if ((handle = open (f_spec, O_BINARY | O_RDONLY)) == -1)
  { return (-1);
  };
  while ((bytes = read (handle, buf, buf_size)) != 0)
  { *crc = block_crc (buf, bytes, *crc);
  };
  close (handle);
  return (0);
};
```

Funktion "Make CRC Table":
`void make_crc_table (WORD polynomial)`

Aufrufparameter:

`polynomial`	Generatorpolynom

Seiteneffekte:

`crc_table`	vorberechnete Daten für `block_crc`

Rückgabewert:
keiner

Tabelle A.29: `make_crc_table`: Berechne CRC-Tabelle

Funktion "Compute CRC-Checksum Over File":
`int sig_crc (BYTE buf[], size_t buf_size, char f_spec[], WORD *crc)`

Aufrufparameter:

`buf`	Adresse Puffer
`buf_size`	Puffergröße in Bytes
`f_spec`	vollständiger Dateiname
`crc`	Adresse Startwert für Prüfsumme

Seiteneffekte:

`*crc`	Prüfsumme über Datei

Rückgabewert:
0: kein Fehler; -1: Datei nicht gefunden

Tabelle A.30: `sig_crc`: Berechne CRC-Prüfsumme über Datei

Hinweis. Die Sicherheit von CRC-Prüfsummen bei der Abwehr von Computerviren und anderen Softwareanomalien wird oft kontrovers diskutiert. Allgemein gilt für Prüfsummen, daß sie den Inhalt einer Datei auf kleinem Platz widerspiegeln sollen. Verschiedene Dateien führen möglichst zu unterschiedlichen Prüfsummen, um Verfälschungen erkennen zu können.

Die Frage nach dem Grad der Sicherheit ist also gleich der Frage, wie schwierig es ist, eine Datei mit beliebigem Inhalt so zu generieren, daß eine vorgegebene Prüfsumme resultiert. Bei CRC-Prüfsummen ist dies bei bekanntem Generatorpolynom recht einfach. Zusätzlich wird oft kritisiert, daß sich die Referenzprüfsummen in einer Datei auf dem Rechner befinden, also im Einzugsbereich der Softwareanomalien. Ein Anwender hat mit diesen Vorgaben wahrscheinlich nur mäßige Schwierigkeiten, eine Datei unsichtbar für das Prüfsummenprogramm zu manipulieren. Trotzdem dürfte es für ein Virus unmöglich sein, das Generatorpolynom und die Tabelle ausfindig zu machen oder die Zieldatei sinnvoll zu verändern. Der Aufwand ist für ein Programm der Größe eines le-

benstauglichen Virus einfach zu groß. Schon durch die Verwendung eines vom Benutzer bestimmten Startwertes wie in unserem Fall wäre das Virus auf Probieren angewiesen.

Kryptographische Prüfsummen, wie sie das MD4RSA-Verfahren und der DES anbieten, können die Durchbruchsicherheit stark erhöhen. Allein schon die Signaturlänge von 16 bzw. 8 Bytes erhöht potentiell den Schutz. Allerdings sind diese Verfahren erheblich rechenaufwendiger, so daß am besten Hardwareimplementationen die Berechnung anstelle der CPU übernehmen.

Check if Device (is_device). Die ANSI-"C"-Funktion `isatty` stellt für ein Handle fest, ob die zugehörige Datei eine normale Datei oder ein Gerät ist. `is_device` übernimmt das probeweise Öffnen und Schließen der zur überprüfenden Datei.

Funktion "Check if Device":
`int is_device (char f_spec[])`

Aufrufparameter:

`f_spec`	Dateispezifikation

Seiteneffekte:
keine

Rückgabewert:
0: Datei oder Datei nicht gefunden; 1: Gerät

Tabelle A.31: `is_device`: Prüfe, ob Dateiname zu einem Gerät gehört

Display Message (message) zeigt eine einzeilige Nachricht auf den Bildschirm an, ohne den aktuellen Inhalt zu zerstören oder komplexe "C"-Funktionen zu benutzen. Die Zeichenein-/-ausgabe erfolgt mit Hilfe der Funktionen `readchar` und `writechar`, die sich wiederum Funktionen des Video-Interrupts bedienen. Ort der Operation ist jeweils die Videoseite 0 und der aktuelle Standort des Cursors. `xwherexy` und `xgotoxy` übernehmen das Abfragen bzw. Setzen der Cursorposition, wobei ebenfalls BIOS-Aufrufe eingesetzt werden.

Funktion. Die aktuelle Cursorposition wird in `old_x` und `old_y` gerettet, der Inhalt der Nachrichtenzeile (Zeichen *und* Attribute) in `old` gespeichert und mit Leerzeichen überschrieben. Alle Ausgaben erfolgen generell mit dem Zeichenattribut `attr`.

```
struct T_CHAR                           /* einfuegen in "Typedefs" */
{ BYTE  chr;
  BYTE  attr;
};

BYTE message (BYTE attr, char text[])
{ int x;                                /* Zaehler                 */
  int old_x, old_y;                     /* alte Cursorposition     */
  BYTE key;                             /* gedrueckte Taste        */
  struct T_CHAR old[80];                /* Zeichenpuffer           */

  /* Cursorposition und Zeileninhalt retten, Zeile loeschen        */
```

```
  xwherexy (&old_x, &old_y);
  for (x = 0; x < 80; x++)
  { xgotoxy   (1 + x, 12);
    readchar  (&old[x].attr, &old[x].chr);
    writechar (attr, ' ');
  };
```

Anschließend wird die Nachricht `text` zentriert ausgegeben und auf einen Tastendruck `key` des Benutzers gewartet, der gleichzeitig Rückgabewert der Funktion ist. Evtl. bereits anstehende Zeichen im Tastaturpuffer werden überlesen.

```
  /* Nachricht ausgeben, auf Tastendruck warten                    */
  x = 40 - (strlen (text) >> 1);
  while (*text)
  { xgotoxy (x++, 12);
    writechar (attr, *(text++));
  };
  while (xbioskey (1))
  { xbioskey (0);
  };
  while (!xbioskey (1));
  key = xbioskey (0);
```

In der Nachbearbeitung wird der Inhalt der Zeile restauriert und der Cursor auf seine alte Position zurückgesetzt.

```
  /* Zeile und Cursorposition restaurieren                         */
  for (x = 0; x < 80; x++)
  { xgotoxy (1 + x, 12);
    writechar (old[x].attr, old[x].chr);
  };
  xgotoxy (old_x, old_y);
  return (key);
};
```

Funktion "Display Message":
`BYTE message (BYTE attr, char text[])`

Aufrufparameter:

`attr`	Attributwert für Textausgabe
`text`	auszugebende Nachricht

Seiteneffekte:
Die Nachricht wird auf dem Bildschirm angezeigt; der Tastaturpuffer wird geleert

Rückgabewert:
gedrückte Taste

Tabelle A.32: `message`: Gebe Nachricht auf Bildschirm aus

Tokenise Rights String (tokenise) wandelt einen Rechtestring `rights` anhand der Tokentabelle `token` in ein Rechtewort um. `token` ist ein bis zu 16 Zeichen langer String, der die Kennbuchstaben der verfügbaren Rechte enthält (z.B.

"`sfi_________trwx`"). Stimmt ein Zeichen in `rights` mit einem Zeichen in `token` überein, wird im Rückgabewert das der Position in `token` entsprechende Bit gesetzt. Das erste Zeichen in `token` entspricht Bit 15, das 16. Zeichen Bit 0. Beispielsweise wäre das Ergebnis für den Rechtestring "`riws`" und die oben angeführte Tokentabelle gleich $C006_{16}$. Die Reihenfolge der Rechte in `rights` ist beliebig, Groß- und Kleinschreibung wird unterschieden.

Funktion "Tokenise Rights String":
`WORD tokenise (char rights[], char token[])`

Aufrufparameter:

`rights`	zu konvertierender Rechtestring (max. 16 Zeichen)
`token`	String mit verfügbaren Rechten (max. 16 Zeichen)

Seiteneffekte:
keine

Rückgabewert:
Ergebnis der Konvertierung

Tabelle A.33: `tokenise`: Wandle Rechtestring in Rechtewort um

A.4.2 Bearbeitung von Dateinamen

Add Startpath To Filename (add_path). Unsere Programme greifen fast alle auf Dateien zurück, aus denen sie für den Betrieb wichtige Informationen beziehen. Die Hilfsdateien befinden sich normalerweise im gleichen Verzeichnis wie die Programme. Nun ist aber das aktuelle Verzeichnis nicht notwendigerweise mit dem Startverzeichnis identisch, wenn das zu startende Programm per Suche über `path` gefunden wurde. Ähnliches gilt für TSR-Programme, die ständig im Hintergrund ablaufen. Die Angabe eines festen Pfades ist uns zu starr, aber Dateizugriffe ohne Pfadangabe beziehen sich stets auf das aktuelle Verzeichnis. Falls dieses nicht das Startverzeichnis ist, laufen Dateizugriffe des Programms ins Leere.

Funktion. Im Argument `argv[0]` von `main` ist der vollständige Programmname, d.h. Dateiname inklusive Startpfad, enthalten. Die Funktion `add_path` macht sich diesen Umstand zu Nutze und stellt einem übergebenen Dateinamen den Startpfad voran. Mit diesem voll qualifizierten Dateinamen kann der Zugriff korrekt und unabhängig vom aktuellen Verzeichnis erfolgen.

Compare Filenames (cmp_fname). Zum Vergleich zweier Dateinamen, die Jokerzeichen, aber keine Pfadangabe enthalten dürfen, dient die Funktion `cmp_fname`. Da der Vergleich von Dateinamen in "konventioneller" Form ("C"-Strings) nicht ganz einfach ist, bedienen wir uns eines Tricks. Die Funktion "Parse Filename" des Kernel erleichtert uns die Sache ungemein, indem sie Namen und Erweiterung in die fixen Felder eines FCB überträgt, leere Felder mit Leerzeichen füllt und das Zeichen '`*`' gegen eine

Funktion "Add Startpath To Filename":
`add_path (char p_spec[], char f_name[], char f_spec[])`

Aufrufparameter:

`p_spec`	vollst. Programmname (`argv[0]`)
`f_name`	zu ergänzender Dateiname
`f_spec`	Adresse Zielstring

Seiteneffekte:

`f_spec`	ergänzter Dateiname

Rückgabewert:
Zeiger auf `f_spec`

Tabelle A.34: `add_path`: Stelle Dateinamen Startpfad voran

äquivalente Folge von '?' ersetzt. So wird der Dateiname "`abc??.*`" zu "`abc??    ???`" und "`*.m?x`" zu "`????????m?x`". Der erste Absatz der Funktion nimmt die Umwandlung der beiden Dateinamen `name` und `mask` in die FCBs `fcb1` und `fcb2` vor.

```
int cmp_fname (char name[], char mask[])
{ struct T_FCB fcb1, fcb2;
  int i, points;

  xparsfnm (name, &fcb1, 0x00);
  xparsfnm (mask, &fcb2, 0x00);
```

Dann erfolgt der eigentliche Vergleich, der, wie die simple `for`-Schleife zeigt, geradlinig und einfach realisiert werden kann. Für jede perfekte Übereinstimmung wird `points` inkrementiert, falls das Zeichen kein Leerzeichen war. Passen die Zeichen nicht zusammen, muß die Maske in `fcb2` ein '?' enthalten, sonst schlägt der Vergleich fehl und die Schleife wird verlassen. Für Jokerzeichen gibt es keine Punkte. Nur, wenn alle 11 Zeichen übereinstimmen, gibt die Funktion den Wert von `points` zurück; sonst signalisiert -1 die Verschiedenheit der beiden Namen. Die Bewertung des Vergleichs spielt für `AVWatchF` (s.a. 4.5.8) eine große Rolle.

```
  points = 0;
  for (i = 0; i < 11; i++)
  { if (fcb1.name[i] == fcb2.name[i])
    { points += (fcb1.name[i] != ' ');
    }
    else
    { if (fcb2.name[i] != '?')
      { break;
      };
    };
  };
  return ((i == 11) ? points : -1);
};
```

Hinweis. Der Rückgabewert 0 bedeutet nicht, daß die Namen nicht überstimmen, sondern daß keine zwei Zeichen tatsächlich gleich waren. Für Treffer mit Jokerzeichen

Funktion "Compare Filenames":
`int cmp_fname (char name[], char mask[])`

Aufrufparameter:

`name`	Dateiname
`mask`	Maske (mit Jokerzeichen)

Seiteneffekte:
keine

Rückgabewert:
-1: keine Übereinstimmung; n: Anzahl der übereinstimmenden Zeichen

Tabelle A.35: `cmp_fname`: Vergleiche Dateinamen

werden nämlich keine Punkte vergeben, da z.B. der Dateiname `*.*` auf alle Dateien paßt. Der durch den Vergleich erzielte Informationsgewinn ist gleich 0. Der Vergleich einer Datei mit einer Maske, die keine Jokerzeichen enthält, verschafft uns ein Maximum an Information: Genau *eine* Datei kann mit der Maske übereinstimmen.

Compose Filename (comp_fspec). Beim Zusammenfügen von Pfad und Namen zu einer vollständigen Dateispezifikation müssen beide Teile durch einen Backslash '\' getrennt werden. Dieser darf aber nur eingefügt werden, wenn der Pfad nicht mit dem Wurzelverzeichnis identisch ist, das immer mit einem Backslash endet. `comp_fspec` berücksichtigt diese Ausnahme.

Funktion "Compose Filename":
`char *comp_fspec (char path[], char f_name[])`

Aufrufparameter:

`path`	Pfad
`f_name`	Dateiname

Seiteneffekte:

path	vollst. Dateispezifikation

Rückgabewert:
Zeiger auf `path`

Tabelle A.36: `comp_fspec`: Füge Pfad und Dateinamen zusammen

Fill In Filename (fill_in). Die DOS-Funktionen `copy` und `rename` akzeptieren Jokerzeichen sowohl in der Quell- als auch in der Zielangabe. Die Abbildung von Dateinamen auf die Zielangabe ist schwierig zu verstehen, wenn man den Trick mit der Funktion "Parse Filename" nicht kennt. Dazu ein paar Beispiele für die Abbildung des Dateinamens `2000.hal` auf verschiedene Zielmasken (Maske links, Ergebnis rechts):

```
dave.bow     →  dave.bow
*.*          →  2000.hal
??1?.s??     →  2010.sal
?????/10.?   →  2000/10.h
```

Falls die Namen als "C"-Strings belassen werden, wird die Abbildung sehr kompliziert. Deshalb verwenden wir das FCB-Format, in dem sich das Beispiel von oben schon durchsichtiger darstellt (links Maske, rechts Ergebnis):

```
2000    hal        (Quelle)

dave    bow    →   dave.bow
??????????     →   2000.hal
??1?    s??    →   2010.sal
?????/10?      →   2000/10.h
```

Funktion. Wir stellen folgendes fest:

1. Def.: Ein Zeichen ist weder Joker- noch Leerzeichen.
2. Zeichen aus der Maske werden übernommen.
3. Für Jokerzeichen werden Zeichen aus der Quellangabe eingesetzt.
4. Leerzeichen werden übersprungen und zählen als bei der Umwandlung in einen "C"-String als Stringende (jeder Namensteil separat).

Genau nach diesen Regeln bearbeitet die Funktion `fill_in` beide Namensteile. Die Behandlung muß separat erfolgen, weil erst nach Einsetzen der Erweiterung feststeht, ob der Zielname überhaupt eine Erweiterung besitzt, die durch einem Punkt vom Namen zu trennen ist. Außerdem ist Punkt 4. zu beachten.

Funktion "Fill In Filename":
`void fill_in (char name[], char mask[], char fill[])`

Aufrufparameter:

`name`	Dateiname (kein Pfad)
`mask`	Maske
`fill`	Zielvariable

Seiteneffekte:

`fill`	Ergebnis der Abbildung

Rückgabewert:
keiner

Tabelle A.37: `fill_in`: Bilde Dateinamen auf Maske ab

Split Filespecification (split_fspec) extrahiert aus einer Dateispezifikation bestimmte Namensteile, die durch den Split-Modus spezifiziert sind:

```
#define SM_DRIVE        0               /* Laufwerk                */
#define SM_PATH         1               /* Pfad (Lfw.+Verzeichnis) */
#define SM_DIRECTORY    2               /* Verzeichnis             */
#define SM_NAME         3               /* Name inkl. Erweiterung  */
#define SM_NAMEPART     4               /* Name                    */
#define SM_EXTPART      5               /* Erweiterung             */
```

Funktion "Split Filespecification":
`char *split_fspec (int split_mode, char f_spec[], char result[])`

Aufrufparameter:

`split_mode`	Zerlegungsmodus (Auswahl Namensteil; s. Text)
`f_spec`	zu zerlegende Dateispezifikation/Pfad
`result`	Zielvariable

Seiteneffekte:

`result`	spezifizierter Namensteil

Rückgabewert:
Zeiger auf `result`

Tabelle A.38: `split_fspec`: Zerlege Dateinamen

A.4.3 Stringverarbeitung

Clean String (clean) bringt einen Textstring in Normalform. Die einzelnen Verarbeitungsoptionen lassen sich mit `OR` kombinieren:

```
/*                                  Entfernen:              */
#define CM_HEAD         0x01        /* Leerzeichen am Anfang  */
#define CM_TAIL         0x02        /* Leerzeichen am Ende    */
#define CM_DOUBLE       0x04        /* mehrfache Leerzeichen  */
#define CM_CTRL         0x08        /* Steuerzeichen          */
/*                                  Umwandeln in:           */
#define CM_LOCASE       0x10        /* Kleinbuchstaben        */
#define CM_HICASE       0x20        /* Grossbuchstaben        */
```

Compare String Until character (stricmpu) funktioniert ähnlich wie die ANSI-Funktion `strcmp`, verfügt aber über den zusätzlichen Parameter `until`. Dieser bestimmt ein Zeichen (sog. Delimiter), bei dem der Vergleichsvorgang wie bei Erreichen des Stringendes abgebrochen wird.

Copy String Until Character (strcpyu) kopiert wie `strcpy` einen String, stoppt aber wie `strcmpu`, falls das Zeichen `until` auftritt.

Search String for String (stristr) arbeitet wie `strstr`, nur daß Unterschiede in der Groß- und Kleinschreibung ignoriert werden.

Funktion "Clean String":
`char *clean (int mode, char text[])`

Aufrufparameter:

`mode`	Modus (s. Text)
`text`	zu bearbeitender String

Seiteneffekte:

`text`	entsprechend `mode` verändert

Rückgabewert:
Zeiger auf `text`

Tabelle A.39: `clean`: Bereinige String

Funktion "Compare Strings Until Character":
`int stricmpu (char a[], char b[], char until)`

Aufrufparameter:

`a`	String 1
`b`	String 2
`until`	Endezeichen

Seiteneffekte:
`keine`

Rückgabewert:
<0: `a` < `b`; 0: `a` = `b`; >0: `a` > `b`

Tabelle A.40: `stricmpu`: Vergleiche Strings bis <Zeichen>

Funktion "Copy String Until Character":
`char *strcpyu (char *dest, char *source, char until)`

Aufrufparameter:

`dest`	Zielstring
`source`	Quellstring
`until`	Endezeichen

Seiteneffekte:

`dest`	`source`

Rückgabewert:
Zeiger auf `dest`

Tabelle A.41: `strcpyu`: Kopiere String bis <Zeichen>

Funktion "Search String":
`char *stristr (char a[], char b[])`

Aufrufparameter:

`a`	zu durchsuchender String
`b`	Suchstring

Seiteneffekte:
keine

Rückgabewert:
`NULL`: nicht gefunden; sonst: Zeiger auf gesuchten String in `a`

Tabelle A.42: `stristr`: Suche String in String

A.4.4 Listenverwaltung

Das Paket `linklist.c` enthält Funktionen zur Verwaltung sequentieller, vorwärts verketteter sortierter Listen. Die Funktion `add2list` fügt einer Liste einen Eintrag hinzu, `delete_list` gibt den Listenspeicher wieder frei. Des weiteren lädt `load_list` eine Textdatei zeilenweise mit `add2list` in eine Liste ein.

Add to List (add2list) reserviert zunächst Platz für einen neuen Knoten vom Typ `struct T_ENTRY`. Dann wird die Liste nach dem ersten Element durchforstet, das größer oder gleich dem einzufügenden Element ist. Der neue Knoten wird vor diesem Element eingefügt (dabei entsteht möglicherweise ein neuer Anker[1]) oder aber an die Liste angehängt, falls das neue Element das größte Element der Liste ist. Für den Text wird Speicherplatz reserviert und der Verweis darauf in `text` eingetragen.

```
struct T_ENTRY
{ char  *text;                          /* Zeiger auf Texteintrag  */
  struct T_ENTRY *next;                 /* Zeiger auf Nachfolger   */
};
```

Delete List (delete_list) durchläuft die Liste und gibt den Speicherplatz für jeden Knoten und die Daten frei.

Load List (load_list) basiert auf `add2list` und fügt alle Zeilen der Datei `name` in die durch `anchor` bezeichnete Liste ein. Kommentarzeilen, die durch ein Prozentzeichen in der ersten Spalte markiert sind, werden überlesen. Mit Hilfe von `clean` werden gleichzeitig alle führenden Leerzeichen und Kontrollzeichen entfernt (Tab. A.39).

Check if Selected (select_p) gibt Auskunft darüber, ob der Dateiname `name`, der Jokerzeichen, aber keine Pfadangabe enthalten darf, in der durch `anchor` bezeichneten Liste enthalten ist.

```
int selectp (struct T_ENTRY *anchor, char name[])
{ while ((anchor != NULL) && (cmp_fname (name, anchor -> text) == -1))
  { anchor = anchor -> next;
```

[1] Zeiger auf ersten Knoten = Wurzelknoten einer Liste.

Funktion "Add To List":
`int add2list (struct T_ENTRY **anchor, char text[]`

Aufrufparameter:

`anchor`	*Adresse* des Zeigers auf erstes Element (Anker)
`text`	einzufügender Text

Seiteneffekte:

`text`	wird in die Liste einsortiert; `anchor` u.U. verändert

Rückgabewert:
0: kein Fehler; -1: zu wenig freier Speicher

Tabelle A.43: `add2list`: Sortiere Text in Liste ein

Funktion "Delete List":
`void delete_list (struct T_ENTRY **anchor)`

Aufrufparameter:

`anchor`	*Adresse* des Zeigers auf erstes Element (Anker)

Seiteneffekte:
Der durch die Liste belegte Speicher wird freigegeben; `*anchor` auf `NULL` gesetzt

Rückgabewert:
keiner

Tabelle A.44: `delete_list`: Gebe durch Liste belegten Speicher frei (lösche Liste)

```
  };
  return (anchor != NULL);
};
```

A.4.5 Zugriff auf die Kommandozeile

Die zwei Routinen der Argument-Gruppe sind in der Datei `chk4args.c` zusammengefaßt. Sie dienen zur Bearbeitung der Kommandozeile, die aus geordneten Textparametern und Schaltern (vorangestelltes '-' oder '/') in beliebiger Reihenfolge besteht. Wie kommt man von "C" aus an die Kommandozeile heran? Prinzipiell besteht wie bei `ReadEnv` die Möglichkeit, mit eigenen Routinen das Environment des Programms zu durchsuchen. "C" (genauer: der Startcode) macht es uns einfacher: Bei der Deklaration von `main` als

```
int main (int argc, char *argv[]);
```

enthält das Array `argv` (**arg**ument **v**alues) Zeiger auf die Parameter der Kommandozeile und `argc` (**arg**ument **c**ount) die Anzahl der Einträge. `argv[0]` ist reserviert

Funktion "Load Textfile In List":
`int load_list (char name[], struct T_ENTRY **anchor)`

Aufrufparameter:

`name`	Name der Textdatei
`anchor`	*Adresse* des Zeigers auf erstes Element (Anker)

Seiteneffekte:
Die Textdatei wird zeilenweise gelesen und in die Liste einsortiert

Rückgabewert:
0: kein Fehler; -1: Datei nicht gefunden; -2: zu wenig freier Speicher

Tabelle A.45: `load_list`: Lese Textdatei zeilenweise in sortierte Liste ein

Funktion "Check if Selected":
`int select_p (struct T_ENTRY *anchor, char f_name[])`

Aufrufparameter:

`anchor`	Zeiger auf erstes Element der Liste (Anker)
`f_name`	Dateiname (kein Pfad)

Seiteneffekte:
keine

Rückgabewert:
0: nicht in Liste; 1: in Liste

Tabelle A.46: `select_p`: Prüfe, ob Dateiname in Liste enthalten ist

und enthält den Namen des Programms inklusive Startpfad. Die folgenden Funktionen erwarten die genannten zwei Angaben und ggf. noch andere Parameter.

Check For Argument (arg_p) prüft das Vorhandensein eines Arguments. Kommt der gesuchte Text `arg` in der Kommandozeile vor (Groß-/Kleinschreibung unerheblich), gibt die Funktion einen von 0 verschiedenen Wert zurück. Der Programmname wird nicht als Bestandteil der Kommandozeile betrachtet.

```
int arg_p (int argc, char *argv[], char arg[])
{ argc--;
  while (argc && stricmp (argv[argc], arg))
  { argc--;
  };
  return (argc != 0);
};
```

Get Argument (get_arg) ermittelt das `nr`-ste Argument der Kommandozeile, wobei Schalter nicht mitgezählt werden. Dadurch dürfen Schalter und Textargumente gemischt auftreten, ohne daß es zu Schwierigkeiten kommt. Zur Funktion: Die Liste der Argumente wird durchlaufen, bis das gesuchte Element gefunden (`count == nr`)

Funktion "Check for Argument":
`int arg_p (int argc, char *argv[], char option[])`

Aufrufparameter:

`argc`	Anzahl der Einträge in `argv`
`argv`	Array mit Zeigern auf die Kommandozeilenparameter
`option`	Adresse des gesuchten Textes

Seiteneffekte:
keine

Rückgabewert:
0: nicht gefunden; 1: gefunden

Tabelle A.47: `arg_p`: Suche Kommandozeilenparameter

oder das Ende der Liste erreicht wird. `count` zählt dabei die Anzahl der Argumente, die keine Schalter sind; `n` dient als Index in `argv`. Die Vorbelegung von `count` mit -1 bewirkt, daß die `while`-Schleife mindestens einmal durchlaufen wird, selbst wenn `nr` = 0 ist. Andernfalls würde in diesem Fall fälschlicherweise überhaupt keine Überprüfung vorgenommen, sondern die Funktion sofort und scheinbar erfolgreich beendet. Vor dem Aussprung aus `get_arg` wird `n` noch korrigiert bzw., falls kein passendes Argument gefunden werden konnte, auf -1 gesetzt.

```
int get_arg (int argc, char *argv[], int nr)
{ int count, n;
  count = -1;
  n     =  1;
  while ((count != nr) && (n < argc))
  { count += ((argv[n][0] != '-') && (argv[n][0] != '/'));
    n++;
  };
  n = (count == nr) ? n-- : 0;
  return (n);
};
```

Funktion "Get Argument":
`int get_arg (int argc, char *argv[], int nr)`

Aufrufparameter:

`argc`	Anzahl der Einträge in `argv`
`argv`	Array mit Zeigern auf die Kommandozeilenparameter
`int nr`	Argumentnummer (Start mit 0)

Seiteneffekte:
keine

Rückgabewert:
Index des gesuchten Arguments in `argv`

Tabelle A.48: `get_arg`: Bestimme Index des n-ten Kommandozeilenparameters

A.4.6 Spezielle Funktionen

Read And Compile Transport Rights (read_t_rights) liest die ASCII-Datei mit den Transportrechten `t_rights.lst` ein, wandelt die Einträge in ein internes Format um und schreibt das Ergebnis in das Array `t_rights` (s.a. 4.4.1 "AVCopy"; `M_DRIVE` hat den Wert 10). Die prinzipielle Funktion ist praktisch identisch mit der von `compile_f_rights` in `AVConfig` (s.S. 231).

Get/Put Reference Entry (get_ref, put_ref). Für den Zugriff auf die von `Chkstate` angelegten Referenzdateien sind die Funktionen `get_ref` (finden, lesen) und `put_ref` (schreiben) zuständig. Die Suche erfolgt in zwei Schritten jeweils sequentiell über die Verzeichnisliste `chkstate.sui` und die Dateiliste `chkstate.fii` (Abb. A.1). Dabei kommt die Funktion `stricmpu` zum Einsatz, die Strings unter Nichtbeachtung der Groß-/Kleinschreibung bis zu einem bestimmten Zeichen oder Textende vergleicht. Dies ist notwendig, weil Namen in den Referenzlisten mit Leerzeichen aufgefüllt sind und nicht nach der signifikanten Information mit einem 0-Byte enden. Ein Vergleich von z.B. "`nelson.bil  `" und "`nelson.bil`" würde sonst negativ ausfallen.

Die Suche nach dem Verzeichnis liefert im Erfolgsfall Informationen über die Lage der zugehörigen Teilliste der Dateinamen (`subdir.start`, `subdir.end`).

Funktion "Read And Compile Transport Rights":
`int read_t_rights (char f_name[], WORD t_rights[M_DRIVE][M_DRIVE])`

Aufrufparameter:

`f_name`	Name der Rechtedatei
`t_rights`	Array für Transportrechte

Seiteneffekte:

`t_rights`	Transportrechte

Rückgabewert:
0: kein Fehler; -1: Rechtedatei nicht gefunden

Tabelle A.49: `read_t_rights`: Lese und kompiliere Transportrechte

Abbildung A.1: Zugriff auf die Referenzliste von `ChkState`

```
int get_ref (char f_spec[], struct T_FILE *file, int *f_nr,
             char f_subdir[], char f_name[])
{ struct T_SUBDIR subdir;              /* Verzeichniseintrag      */
  int  handle;                         /* Datei-Handle            */
  char path[65], name[13];             /* Pfad und Name F_SPEC    */
  char flag;                           /* "gefunden"-Flag         */

  if ((handle = open (f_subdir, O_BINARY | O_RDONLY)) == -1)
  { return (-1);
  };
  split_fspec (SM_PATH, f_spec, path);
  while (read (handle, &subdir, sizeof (struct T_SUBDIR)) != -1)
  { if ((flag = stricmpu (subdir.name, path, ' ')) == 0)
    { break;
    };
  };
  close (handle);
```

```
  if (flag)
  { return (-3);
  };
```

Diese wird ebenfalls sequentiell nach einem passenden Eintrag durchsucht. Zurückgeliefert werden der ausgefüllte Referenzeintrag `*file` und dessen Nummer `*f_nr`, die für den Aufruf der Funktion `put_ref` von Bedeutung ist.

```
  if ((handle = open (f_name, O_BINARY | O_RDONLY)) == -1)
  { return (-2);
  };
  split_fspec (SM_NAME, f_spec, name);
  *f_nr = subdir.start;
  lseek (handle, *f_nr * sizeof (struct T_FILE), SEEK_SET);
  while (*f_nr < subdir.end)
  { if (read (handle, file, sizeof (struct T_FILE)) !=
       sizeof (struct T_FILE))
    { return (-5);
    };
    if ((flag = stricmpu (file -> name, name, ' ')) == 0)
    { break;
    };
    (*f_nr)++;
  };
  close (handle);

  return ((flag) ? -4 : 0);
};
```

Hinweis. Da die Referenzeinträge alphabetisch sortiert vorliegen, bietet sich die binäre Suche als schneller Zugriffsalgorithmus an. Tests des Autors haben gezeigt, daß der Implementationsaufwand (residentes Programm!) in keinem Verhältnis zum erzielten Geschwindigkeitsgewinn steht. "Schuld" daran sind die relativ kurzen Listen, die zu durchsuchen sind. Die Liste der Verzeichnisse umfaßt i.d.R. wohl weniger als 40–50 Einträge; ähnliches dürfte für die Teillisten der Dateinamen gelten. Bei der sequentiellen Suche sind demnach im Durchschnitt etwa 20–25 Suchschritte notwendig, bei der binären etwa $\log_2 50 \approx 6$. Das entspricht ca. $\frac{1}{3}$ bis $\frac{1}{4}$ der Zugriffszeit beim sequentiellen Verfahren; ein Verhältnis, das sich für mehr Einträge schnell verbessert.

Ob das viel ist oder nicht, hängt von den eingesetzten Programmen ab. Bei der Textverarbeitung oder Programmerstellung werden nur mäßig häufig Dateien geöffnet und bearbeitet. Ganz anders die Sachlage bei Datenbanken: Ständig findet eine Vielzahl von Dateioperationen statt, deren Geschwindigkeit starken Einfluß auf die gesamte Leistung des Programms hat. Andererseits sollte ein TSR-Programm möglichst kurz gehalten sein, und dazu tragen simple Algorithmen bei. Im Einzelfall ist also abzuwägen, ob Geschwindigkeit oder verfügbarer Hauptspeicher Optimierungsziel sind.

Das Gegenstück zu `get_ref`, `put_ref`, ist erheblich simpler aufgebaut. Das liegt daran, daß kein Eintrag mehr gesucht werden muß, sondern die Nummer direkt mit `f_nr` übergeben wird. `*file` enthält die in die Datei `f_file` zu schreibenden Daten.

```
int put_ref (struct T_FILE *file, int f_nr, char f_name[])
{ int handle;                          /* Handle Dateiliste       */

  if ((handle = open (f_name, O_BINARY | O_WRONLY)) == -1)
```

Funktion "Get File Reference Entry":
`int get_ref (char f_spec[], struct T_FILE *file, int *f_nr, char f_subdir[], char f_file[])`

Aufrufparameter:

`f_spec`	vollst. Name der gesuchten Datei
`file`	Adresse zu lesende Daten
`f_subdir`	Name der Referenzdatei (Verzeichnisliste)
`f_nr`	Zeiger auf Eintragsnummer
`f_name`	Name der Referenzdatei (Dateiliste)

Seiteneffekte:

`file`	gelesene Daten
`f_nr`	Eintragsnummer

Rückgabewert:
0: kein Fehler; -1: Verzeichnisliste nicht gefunden; -2: Dateiliste nicht gefunden; -3: Verzeichnis nicht gefunden; -4: Datei nicht gefunden; -5: Lesefehler

Tabelle A.50: `get_ref`: Suche und lese Dateieintrag in Referenzdatei

```
  { return (-1);
  };
  /* auf Eintrag positionieren, Eintrag schreiben                    */
  lseek (handle, f_nr * sizeof (struct T_FILE), SEEK_SET);
  if (write (handle, file, sizeof (struct T_FILE)) == -1)
  { return (-2);
  };
  return (0);
};
```

Funktion "Put File Reference Entry":
`int put_ref (struct T_FILE *file, int f_nr, char f_file[])`

Aufrufparameter:

`file`	Adresse zu schreibende Daten
`f_nr`	Eintragsnummer
`f_file`	Name der Referenzdatei (Dateiliste)

Seiteneffekte:
der Eintrag wird in die Referenzdatei geschrieben

Rückgabewert:
0: kein Fehler; -1: Dateiliste nicht gefunden; -2: Schreibfehler

Tabelle A.51: `put_ref`: Schreibe Dateieintrag in Referenzdatei

Get File Rights (get_rights) stellt für eine Subjekt – Operation – Objekt –

Kombination anhand einer Rechtedatei (normalerweise "`f_rights.lst`") fest, ob die angeforderte Operation zulässig ist oder nicht. Der vollständige Name der Rechtedatei `f_rights` muß vom aufrufenden Programm definiert werden. Wegen der Möglichkeit, daß auf eine Datei mehrere Einträge zutreffen, muß die gesamte Rechtedatei nach dem am besten passenden Eintrag durchsucht werden (Format s. 4.5.10 "AVConfig", Funktion `compile_f_rights`).

```
/* Name der Rechtedatei (ist im aufrufenden Programm definiert)    */
extern char f_rights[];

int get_rights (char subject[], WORD *rights, char object[], char def)
{ struct T_FR_ENTRY r_entry;            /* Eintrag fuer Dateirechte*/
  WORD hit_subj, hit_obj;               /* Trefferpunkte (TP)      */
  WORD best_rights;                     /* Rechte bester Treffer   */
  int current, best;                    /* TP letzter/bester Eintr.*/
  int handle;                           /* Datei-Handle            */
  char valid;                           /* Bewertung bester Treffer*/
  char result;                          /* Bewertung letzt. Treffer*/

  /* oeffne Rechtedatei                                            */
  if ((handle = open (f_rights, O_RDONLY | O_BINARY)) == -1)
  { message (0xCF, "Couldn't open rights list");
    return (-1);
  };
```

`hit_subj` und `hit_obj` halten das Vergleichsergebnis für Subjekt bzw. Objekt fest. Falls Subjekt oder Objekt nicht übereinstimmen, geht es weiter zum nächsten Eintrag. Falls der Grad der Übereinstimmung das bisherige Maximum `best` erreicht oder überschreitet, wird `result` das Ergebnis der Zulässigkeitsüberprüfung zugewiesen. Zur Erläuterung: Ein erforderliches Recht ist durch ein gesetztes Bit in `*rights` markiert. Falls eine Operation nicht zulässig ist (Bit in `r_entry.rights` ist 0), wird durch die `AND`-Verknüpfung das korrespondierende Bit im Ergebnis gelöscht.

```
  /* suche nach am besten passenden Eintrag                        */
  best   = -1;
  result = -1;
  valid  = def;
  while (read (handle, &r_entry, sizeof (struct T_FR_ENTRY)) ==
      sizeof (struct T_FR_ENTRY))
  { /* passt?                                                      */
    hit_subj = cmp_fspec (FM_LAX, subject, r_entry.subject);
    hit_obj  = cmp_fspec (FM_LAX, object,  r_entry.object);
    if ((hit_subj != 0xFFFF) && (hit_obj != 0xFFFF))
    { /* besser oder gleich gut?                                   */
      if ((current = (hit_subj + hit_obj)) >= best)
      { result = (*rights == (*rights & r_entry.rights));
```

War der Grad der Übereinstimmung höher als das bisherige Maximum, nimmt `valid` den Wert von `result` an. Bei gleich guter Übereinstimmung entscheidet der Wert von `def`.

```
        if (current > best)             /* besser?                 */
        { best_rights = r_entry.rights;
          best = current;
          valid = result;
        }
        else                            /* gleich gut!             */
```

```
            { if (valid != result)
              { valid = def;
              };
            };
          };
        };
      };
      close (handle);

      /* gebe Rechte des besten Treffers zurueck                    */
      *rights = best_rights;
      return ((result != -1) ? valid : -2);
    };
```

Funktion "Get File Rights":
`int get_rights (char subject[], WORD *rights, char object[], char def)`

Aufrufparameter:

`subject`	Subjekt (Name anforderndes Programm)
`rights`	Adresse erforderliche Rechte
`object`	Objekt (betroffene Datei)
`def`	Voreinstellung (falls kein Treffer/mehrere gleich gute Treffer)

Seiteneffekte:

`*rights`	gewährte Rechte

Rückgabewert:
1: zulässig; 0: unzulässig; -1: Lesefehler; -2: kein Eintrag gefunden

Tabelle A.52: `get_rights`: Überprüfe Rechtmäßigkeit einer Operation

Anhang B

Einführung in die Benutzung öffentlicher Datennetze

Kein Medium außer den Computernetzen kann mit der Geschwindigkeit mithalten, mit der neue Softwareanomalien geschrieben werden und sich verbreiten. Damit stellt die elektronische Kommunikation die beste Möglichkeit dar, viele Teilnehmer auf der ganzen Welt schnell und umfassend mit Warnungen und Tips zur Bekämpfung zu versorgen. Im Falle des WANK-Wurms dauerte es nur Stunden von der ersten Entdeckung bis zur Alarmmeldung und Verbreitung von Programmen zur Abwehr. Die Ausbreitung konnte so im Keim erstickt werden.

Es soll hier nicht verschwiegen werden, daß ein Computernetz überhaupt erst die Existenz des Wurms ermöglicht hat. Auch dürfte mancher Trojaner oder Virus von Netzen profitiert haben, sei es durch aktive Benutzung oder passiv durch die Versendung verseuchter Programme. Durch meist verantwortungsvolle Netz- und Mailboxbetreiber kam es bis jetzt noch nicht zur einer massenhaften Verbreitung von Softwareanomalien, von den Wurm-Epidemien einmal abgesehen. Insgesamt wiegen die Vorteile des elektronischen Datenaustauschs die Nachteile allemal auf.

Öffentliche Netze wie BITNET, INTERNET und EARN sind laut Satzung dafür da, daß Wissenschaftler schnell und unkompliziert Forschungsergebnisse miteinander austauschen können. In fast keiner anderen Disziplin dürfte die Umsetzung dieses Gedankens so gelungen sein wie bei der Bekämpfung von Softwareanomalien. Experten und Laien aus aller Welt konferieren über neue Viren, Abwehrmaßnahmen, theoretische und philosophische Aspekte. Dokumente und Programme sind für jedermann mit Netzzugriff verfügbar. Die folgenden Abschnitte erläutern Grundprinzipien der Nachrichtenübermittlung in Computernetzen, deren Benutzung und den Zugriff auf diverse Server- und Informationsdienste.

B.1 Grundlagen Electronic Mail

Eine über ein Computernetz versandte Nachricht ist im Prinzip eine Datei, die analog zu einem Brief die zu übermittelnde Information und Zusatzdaten wie Zieladresse und Beförderungshinweise enthält. Es gibt zwei grundsätzliche Kommunikationsformen: Interaktive Kommunikation und die Kommunikation über Nachrichtendateien.

Bei der *interaktiven Kommunikation* steht der Anwender direkt mit dem Zielrechner (engl. remote host) in Kontakt. Durch die Dialogverbindung ist das Ergebnis jeder Eingabe und der dadurch ausgelösten Aktion unmittelbar sichtbar; jedenfalls so unmittelbar, wie es die Belastung der Rechner und des Netzes zuläßt. Falls Dateien bestellt werden, sendet der Rechner meist nur eine Auftragsbestätigung zurück. Der Versand der Dateien erfolgt oft zeitversetzt, um die Belastung des Rechners zu verteilen und billige Nachttarife zu nutzen.

Die *Kommunikation über Nachrichtendateien* entspricht der Stapelverarbeitung von Informationen (z.B. `bat`-Dateien unter MS-DOS). Der Benutzer versendet eine Datei, die eine ganze Reihe von Anweisungen enthalten kann. Der Zielrechner verarbeitet diese irgendwann und sendet die Ergebnisse als Datei zurück. Bei manchen Servern bilden sich lange Warteschlangen, und es kann ein paar Tage dauern, bevor der Auftrag auch nur bearbeitet wird. Besonders bei fehlerhaften Aufträgen kann das sehr ärgerlich sein. Komfortablere Server prüfen die Bestellung vorab und schicken zumindest eine Auftragsbestätigung, evtl. mit Hinweisen zum Verfahren und zur Serverbelastung, zurück (z.B. `trickle`).

Je nach Netz, Betriebssystem und installierter Software stehen Ihnen eine oder beide Kommunikationsformen zur Verfügung. Erkundigen Sie sich im Zweifelsfall bei Ihrem Systemadministrator über die entsprechenden Kommandos zum Versenden einer Nachricht. Nutzen sie Hilfeprogramme des Rechners und Hilfeoptionen des Kommunikationsprogramms, die oft und reichlich vorhanden sind.

Unabhängig von der Kommunikationsform wird eine Nachricht elektronisch übermittelt, indem sie einem Sendeprogramm mit Angabe der Zieladresse übergeben wird. Die Nachricht wird dann, meist über mehrere miteinander verbundene Rechner (*Netzknoten*), zum Zielrechner geschickt, der diese dann an den lokalen Benutzer weiterleitet. Eine Netzadresse umfaßt also zwei Angaben: den Zielrechner und den lokalen Benutzer. Das hat den Vorteil, daß einem Netzknoten nur der Zielrechner bekannt sein muß und daß Benutzernamen nur auf lokalen Rechnern eindeutig sein müssen. Der lokale Benutzer kann auch ein Programm (ein sog. “Server”) sein, der Nachrichten in einem bestimmten Format empfängt, interpretiert, bearbeitet und beantwortet.

Beispiel “Netzadresse”
Die BITNET-Adresse des lokalen Benutzers `frog` an der Universität Giessen (`dgihrz01`) ist `frog@dgihrz01`. Das Zeichen “@” (für “kaufmännisches und”) hat dabei die Bedeutung “at”. Die Adresse liest sich demnach “`frog` at `dgihrz01`”[1].

Beispiel “interaktive Kommunikation”
Sie möchten vom Server `bitftp` auf dem Knoten `pucc` Hilfe anfordern. Unter manchen

[1] Diese ehemalige Adresse des Autors ist *nicht* mehr gültig.

Betriebssystemen sind Kommunikationsfunktionen fest vorgesehen (s. Beispiele). Wo dem nicht so ist, variiert das genaue Verfahren stark, da es eine große Anzahl von Mail-Programmen gibt.

Betriebssystem	Kommando
IBM VM/CMS	`TELL bitftp AT pucc help`
IBM VM/TSO und MVS	`TRANSMIT pucc.bitftp 'help' NOPROLOG`
VAX VMS	`SEND bitftp@pucc help`

Beispiel "Kommunikation per Nachrichtendatei"
Erstellen Sie eine Textdatei (hier "`mail`"), die nichts als das Wort `help` enthält und schicken Sie diese an `bitftp@pucc`.

Betriebssystem	Kommando
IBM VM/CMS	`SENDFILE mail A bitftp AT pucc`
IBM VM/TSO und MVS	`TRANSMIT pucc.bitftp DATASET(mail)`

Eine Nachricht läuft i.d.R. über mehrere Zwischenrechner. Ankommende Nachrichten werden in eine Warteschlange (engl. wait queue) eingereiht und, je nach Bestimmungsort, in die entsprechende Ausgangswarteschlange gestellt. Wie schnell und wann das erfolgt, hängt von der Netzbelastung und der lokal verfolgten Rechenzentrumspolitik ab. Manche Rechner bearbeiten Nachrichten nur in den Nachtstunden, wenn die Rechenlast gering ist. Das bedeutet, daß Rechner in anderen Zeitzonen u.U. zu anderen Zeiten aktiv sind, als die vor- oder nachgeordneten Knoten, was zu zusätzlichen Verzögerungen führt.

Bei ungünstigen Kombinationen von Netzknoten und starker Belastung des Netzes kann es manchmal Tage dauern, bis eine Nachricht ihr Ziel erreicht. Da die meisten Netze von Universitäten und Konzernen betrieben werden, gibt es keine Garantien dafür, daß die Nachricht in einem bestimmten Zeitraum oder überhaupt zugestellt wird. Aber so schlimm ist es meistens nicht. Während der europäischen Morgenstunden herrscht z.B. in den USA noch tiefe Nacht, und die Rechner reagieren sehr schnell. An guten Tagen kann eine Nachricht von 100 kB in zwei Minuten nach Aufgabe der Bestellung bereits eingetroffen sein, obwohl sie vielleicht in jeder Richtung über ein Dutzend Rechner und einen Satelliten gelaufen ist!

B.2 Besonderheiten (UUENCODE)

Nicht alle Mail-Programme sind in der Lage, sog. *Binär*-Dateien zu bearbeiten und weiterzuleiten. Binärdateien sind z.B. Programme, Grafiken und gepackte Dateien, deren Bytes alle der 256 möglichen Werte annehmen können. Wenn das Mail-Programm für Binärdateien ausgelegt ist, entstehen kein Probleme; die Übertragung erfolgt *transparent*. Ein weniger geeignetes Mail-Programm verarbeitet die Binärdatei u.U. als ASCII-Text, interpretiert deshalb bestimmte Codewörter als Steuersequenzen und verfälscht so die Nachricht. Dies erkennen Sie spätestens daran, daß ein übertragenes Programm zu kurz ist, sich nicht entpacken läßt oder bei der Ausführung abstürzt.

Wie können dennoch Binärdateien über beliebige Knoten übertragen werden? Gemeinsamer Nenner aller Mail-Programme ist die korrekte Übermittlung von reinem ASCII-Text. Gesucht ist also ein Verfahren, daß den Wertebereich eines Bytes von 0 – 255 (256 Werte) auf ASCII-Zeichen abbildet, die keine Steuerzeichen sind. Man hat dazu die 64 Zeichen mit den Codes von 32 (Leerzeichen) bis 93 (Apostroph) ausgewählt. Eine solche Umsetzung leistet das weit verbreitete Programm `uuencode`. Drei Bytes des Quellcodes mit jeweils 8 Bits werden in vier Bytes des Zielcodes mit jeweils 6 Bits transformiert. Das dies funktioniert, zeigt eine kleine Rechnung: $3 * 8$ Bits $= 4 * 6$ Bits $= 24$ Bits. Das Ergebnis der Umwandlung trägt normalerweise die Endung `uue`. Durch diese Behandlung bläht sich jede Datei um den Faktor $\frac{4}{3} \approx 1.33$ auf.

Die Umcodierung in nur 64 Zeichen ist bei weitem nicht optimal. Mittlerweile sind auch verbesserte Programm wie `xxencode` verfügbar, die aber auf Servern noch nicht in breiter Front eingesetzt werden. Damit sich die Katze bei der Bestellung der `uu??code`-Programme nicht in den Schwanz beißt, sind diese als Quelltext in Pascal und "C" erhältlich, der sich problemlos verschicken läßt. Wer noch keine Kodier- und Packprogramme besitzt, sollte sich in den Verzeichnissen `<MSDOS.STARTER>` (Codierprogramme, Hilfetexte) und `<MSDOS.ARC-LBR>` (Packprogramme) auf den `trickle`-Servern umschauen (s.a. B.4).

Um Übertragungskosten einzusparen, werden Dateien vor dem Versand zumeist komprimiert. Momentan am gebräuchlichsten ist das Pack-Programm `pkzip`, das auf vielen Servern bereit gehalten wird und Dateien sehr schnell auf ein Mindestmaß reduziert. Kompressionsraten um 30% bei Programmen, 50 – 60% bei Texten und >90% (sic) bei Grafiken sind durchaus realistisch. Die gepackten Dateien tragen die Endung `zip` und werden durch `pkunzip` wieder entpackt.

Beispiel "Senden und Empfangen von Dateien"
Alle Dateien auf Laufwerk `A:` sollen versandt werden. Um Platz zu sparen und nur eine Datei verschicken zu müssen, werden zuvor alle Dateien in `avsys.zip` zusammengefaßt und komprimiert (Operation "**a**dd"):

```
pkzip -a avsys a:\*.*<CR>
<Meldungen des Programms>
```

Für den Versand ist `avsys.zip` noch in eine verträgliche Form umzuwandeln:

```
uuencode avsys.zip<CR>
<Meldungen des Programms>
dir<CR>
avsys.zip
avsys.uue
```

Die Rückübersetzung beim Empfänger erfolgt mit dem Programm `uudecode`. Nach dem Umcodieren liegt wieder die ursprüngliche Datei vor, die meist noch zu entpacken ist.

```
uudecode avsys<CR>
<Meldungen des Programms>
pkunzip avsys<CR>
<Meldungen des Programms>
dir<CR>
avsys.uue
avsys.zip
<alle in avsys.zip enthaltene Dateien>
```

Tabelle B.1 gibt eine Übersicht über die Endungen gepackter Dateien und die zugehörigen Pack- bzw. Entpackprogramme. Ebenfalls in die Übersicht aufgenommen wurden Programme zur Codewandlung. Falls kein Programmpaar zum Packen und Entpacken angegeben ist, werden die entsprechenden Funktionen über Aufrufparameter ausgewählt. Unter anderen Betriebssystemen als MS-DOS existiert eine Vielzahl von Archivierungs- und Kompressionsprogrammen, die z.T. von Namen und Funktion her etwa gleich denen der Tabelle sind. Zwei bedeutende UNIX-Programme wurden schon der Tabelle hinzugefügt. Die Beschreibung der Aufrufparameter ist hier nicht möglich[2], aber fast jedes Programm gibt bei Aufruf ohne Parameter einen kurzen Hilfetext aus.

Endung	Pack-/Entpackprogramm
`arc`	`arc`, `pkpak/pkunpak; pkxarc`
`lhz`	`lharc`, `lha`
`tar`	`tar` (UNIX)
`uue`	`uuencode/uudecode`
`xxe`	`xxencode/xxdecode`
`z`	`compress` (UNIX)
`zip`	`pkzip/pkunzip`
`zoo`	`zoo`

Tabelle B.1: Dateiendungen und zugehörige Pack-/Entpackprogramme

Wir kommen nun zur Anwendung des Besprochenen und schreiten zur Durchforschung von Netzen und Servern, zumindest auf dem Papier. Die folgenden Abschnitte beschreiben Informationsquellen, die nach steigender Komplexität der Benutzung geordnet sind. Besonders empfohlen seien die Diskussionslisten auf `listserv`ern, die auch dem Anfänger schnell zugänglich sind und eine unerschöpfliche Informationsquelle darstellen.

B.3 LISTSERV-Server (Diskussionslisten)

Eine *Diskussionsliste* ist eine Liste von Netzadressen, die von einem Programm, z.B. `listserv` auf `bitnet`-Knoten, verwaltet wird. An einem bestimmten Thema interessierte Benutzer können sich in eine solche Liste eintragen, in dem sie eine Nachricht an das Verwaltungsprogramm schicken. Eine Nachricht an die Liste wird automatisch an alle Teilnehmer weiterversandt. Dabei ist als lokaler Empfänger der Name der Liste, nicht "`listserv`" anzugeben (ein häufig gemachter Fehler).

Beispiel "Eintrag in LISTSERV Diskussionsliste"
Sie möchten sich in die (wärmstens empfohlene!) Virus-Diskussionsliste eintragen. Aus geeigneten Quellen (s. `netserv`) wissen Sie, daß die Liste `virus-l` heißt, auf dem `bitnet`-Knoten `lehiibm1` zu Hause ist und von einem `listserv`-Programm verwaltet wird. Sie müssen nun die Nachricht "`subscribe virus-l <name>`" an `listserv@-`

[2]Die Anleitung zu z.B. `pkzip` hat mehr als 100 Seiten.

lehiibm1 schicken. Unter <name> ist ein Name anzugeben, der mindestens ein Leerzeichen enthält, z.B. "Harrison Ford". Unter diesem Namen ist ein Listenteilnehmer den anderen bekannt. Die Verwendung von Phantasienamen ist möglich, aber nicht üblich. Die Netzwerkadresse des Absenders kennt listserv aus dem automatisch erzeugten Kopf der Nachricht. listserv schickt Ihnen ein Protokoll über Erfolg oder Nichterfolg des Auftrages.

listserver verwalten nicht nur Listen, sondern auch Dateien und Datenbanken. Für den integrierten Fileserver gelten die gleichen Befehle, die auch netserv (s. dort) verwendet. Genauere Informationen zu speziellen listserv-Fähigkeiten liefert das info-Kommando mit der entsprechenden Themenangabe. Eine Liste der Themen kann mit "info ?" bestellt werden. Tabelle B.2 gibt die gebräuchlichsten listserv-Kommandos an. Groß- und Kleinschreibung wird nicht unterschieden; kleingeschriebene Teile der Befehlsnamen können weggelassen werden (z.B. sub statt subscribe).

Syntax	Bedeutung
HELP	Hilfe anfordern (diese Übersicht)
Info {<Thema>|?}	Informationen anfordern (Thema "ref" wie reference empfehlenswert)
INDex <Liste>	Dateiverzeichnis zur Liste anfordern (oft ältere Ausgaben)
List {short|long}	Beschreibung aller Listen dieses Servers anfordern
REView <Liste>	Liste der Teilnehmer anfordern
SIGNOFF	s. UNSubscribe
SUBscribe <Liste> <Name>	sich in Liste eintragen
UNSubscribe <Liste>	sich aus Liste austragen

Tabelle B.2: Befehlsauswahl listserv

B.4 TRICKLE-Server

trickle-Server haben die Aufgabe, Dateien eines ferner Servers lokal weiterzuvermitteln. So würde es z.B. das Netzwerk stark belasten, wenn jeder Benutzer in Europa direkt auf Dateien in den USA zugreifen würde. Das wäre besonders dann unsinnig, wenn viele die gleiche Datei benötigen. Für den sehr großen Server wsmr-simtel20.army.mil in den USA wurden deshalb lokale Vermittlungsstellen wie z.B. trickle@ds0rus1i in Stuttgart eingerichtet. Bestellt ein Benutzer eine Datei, die trickle nicht vorrätig hat, fragt der angeschriebene Server zunächst bei anderen trickles in seiner Umgebung nach. Wenn einer von diesen die Datei im Bestand hat, reicht trickle die Bestellung stellvertretend für den Benutzer weiter. Falls es gar nicht anders geht, holt trickle die Datei aus den USA. Alle weiteren Anfragen können dann lokal bedient werden; die Verbindung nach Übersee wird somit geschont.

Beispiel "Verzeichnisse und Dateien von TRICKLE ordern"
Sie möchten sich von trickle@ds0rus1i den Inhalt des Verzeichnisses <MSDOS.-

`TROJAN-PRO>` (Antivirusprogramme unter MS-DOS; sehr empfohlen) ausgeben lassen und die Datei `00-INDEX.TXT` in diesem Verzeichnis bestellen. Eine Datei dieses Namens existiert in den meisten Verzeichnissen und gibt weitere Informationen zu den Dateien. Groß- und Kleinschrift wird nicht unterschieden. Die entsprechende Nachricht an `trickle@ds0rus1i` lautet :

Kommando	Kommentar
`/pddir <msdos.trojan-pro>`	Inhaltsverzeichnis ausgeben lassen
`/pdget <msdos.trojan-pro>00-index.txt (uue`	Datei holen

Besondere Hinweise. Die Angabe von "`(uue`" bewirkt, daß die Datei vor dem Versenden mit dem Programm `uuencode` umkodiert wird. Zur Entschlüsselung benötigen Sie das Programm `uudecode`, das als "C"- oder Pascal-Quelltext ebenfalls von `trickle` erhältlich ist.

Syntax	Bedeutung
`/HELP`	Hilfetext anfordern (erklärt die folgenden Befehle)
`/PDDIR [<Verzeichnis>]`	Inhaltsverzeichnis ausgeben lassen
`/PDGET <vollst. Dateiname>`	Datei bestellen
`/SUB <Verzeichnis>`	Verzeichnis abonnieren (neue Dateien werden automatisch per Mail angekündigt)
`/UNSUB <Verzeichnis>`	Abonnement kündigen
`/NEW <Verzeichnis> <d>`	alle Dateien anzeigen, die in den letzten `d` Tagen eingetroffen sind

Tabelle B.3: Befehlsauswahl `trickle`

B.5 FTP-Server

FTP steht für das **F**ile **T**ransfer **P**rotocol, das unter vielen Betriebssystemen zur Verfügung steht. Rechner auf der ganzen Welt, die öffentlichen Zugang erlauben, sind über das FTP erreichbar und stellen so vielleicht das größte Reservoir für computerbezogene und sonstige Informationen dar. Um mit einem Host via FTP zu kommunizieren, muß sowohl der entfernte als auch der lokale Rechner über FTP-Dienste und eine FTP-fähige Verbindung (interaktive Kommunikation) verfügen. Der Aufruf der Dienste erfolgt über das Kommando `ftp`.

Verfügt ihr Rechner über keinen FTP-Service oder die Möglichkeit, interaktiv mit anderen Rechnern zu kommunizieren, können Sie die Dienste des FTP-Servers `bitftp@pucc` in Anspruch nehmen. Dieser akzeptiert Eingaben in Form von Dateien, interpretiert darin enthaltene Kommandos, führt die interaktive Sitzung mit dem Zielrechner durch und sendet Ergebnisse wieder per Datei an den Auftraggeber zurück. Ein Nachteil ist, daß dieser Server der einzige seiner Art und in den USA installiert ist.

Dadurch ist der Andrang meist sehr groß. Außerdem gehen Anforderungen für einen Rechner in z.B. Skandinavien zunächst nach Amerika und dann zurück nach Europa. Weil vormittags in Europa aufgegebene Nachrichten die USA in den Nachtstunden erreichen, wird das Problem der Belastung etwas gemildert. Im Frühjahr 1991 mußte man etwa einen Tag auf die vollständige Erledigung seines Auftrags warten.

Beispiel "Verzeichnisse und Dateien per FTP ordern"
Sie möchten sich vom Host `cert.sei.cmu.edu` (eine `internet`-Adresse[3]) den Inhalt des Verzeichnisses `pub` ausgeben lassen (existiert meistens) und die Datei `readme` in diesem Verzeichnis im Binärmodus holen. Bei den Kommandos wird Groß- und Kleinschrift nicht unterschieden, wohl aber bei den Pfad- und Dateinamen, falls Sie sich auf einem UNIX-System befinden. Die entsprechende Nachricht an `bitftp@pucc`, die gleichzeitig als Befehlsübersicht dienen soll, lautet:

Kommando	Kommentar
`FTP cert.sei.cmu.edu`	Angabe des FTP-Hosts
`USER anonymous`	anonymous ist die Gast-Kennung[4]
`CD /pub`	Wechsel ins Verzeichnis `pub`...
`dir`	...und ausgeben lassen
`BINARY`	Binärmodus einschalten (Gegenteil: `ASCII`)
`GET readme`	Datei holen
`QUIT`	Dialog beenden

B.6 NETSERV-Server

`netserv`er sind weder besonders komplex noch schwer zu bedienen. Sie stehen nur deshalb am Schluß dieser Einführung, weil sie lediglich Hilfsdienste zu verschiedenen Netzwerken anbieten. Dies sind Fileserver-Dienste für Informationstexte zu verschiedenen Netzen, Netzübergängen (engl. gateways), Listen von Knoten auf anderen Netzen und Hilfsprogramme zur Netz-Verwaltung (nur für Mainframes).

Beispiel "Hilfe von NETSERV anfordern"
Um einen Hilfetext anzufordern, schickt man das Kommando `help` an `netserv`. Jedes europäische Land besitzt einen zentralen EARN-Knoten, der `netserv`-Dienste anbietet. Dessen Name ergibt sich aus der Länderkennung für Deutschland `D` und dem Wort `EARN`, was zur Netzwerkadresse `netserv@dearn` führt. Für Griechenland oder die Türkei lautet das Ergebnis entsprechend `grearn` bzw. `trearn`. Die Bestellung `get netserv helpfile` sei empfohlen.

B.7 Mailboxen

In Tabelle B.5 finden sich Netzwerkadressen und Anschlußnummern einiger Institute und Privatleute. Das Verfahren bei den durch '@' aufgeteilten BITNET- und INTERNET-

[3] **C**omputer **E**mergency **R**esponse **T**eam: Eingreiftruppe des BITNET.
[4] Der Benutzer `anonymous` benötigt kein Paßwort.

Syntax	Bedeutung
`HELP [<Thema>]`	Hilfe (zu einem bestimmten Thema) anfordern
`GET <Dateiname> <Dateityp>`	Datei bestellen
`Query CMD`	Liste aller Kommandos ausgeben

Tabelle B.4: Befehlsauswahl `netserv`

Adressen ist bereits bekannt. Letztere haben i.d.R. mehr als einen Dezimalpunkt im Adressteil und enden oft mit `edu` (education: Universitäten, Schulen), `com` (commerce: Unternehmen), `gov` (government: Regierung) oder `mil` (military: Militär). Auch X.400-Adressen sind durch Punkte in Felder unterteilt; der letzte Eintrag enthält die Länderkennung, z.B. `de` für Deutschland.

Für die Anschlußnummern ist ein Akustik-Modem oder ein fester DATEX-P-Anschluß und ein entsprechendes Kommunikationsprogramm erforderlich. Für diese Art der Kommunikation sei allerdings auf entsprechende Publikationen verwiesen.

Name	Mail-Adresse
Chip Viren-Service	`fgpc-gm@infohh.rmi.de`
Fridrik Skulason "`F-Prot`"	`frisk@rhi.hi.is`
Micro-BIT Virus Center	`ry15@dkauni11.bitnet`
John McAfee (Alan J. Roberts) "`ViruScan`"	`cup.portal.com`
Viren Test Zentrum Hamburg	`vir-sfh@infohh.rmi.de`
INFO.box	0 40/69 40 101 (300 Bd 7E1)
Login-name: "fgpc"	0 40/69 40 145 (1.2 kBd, 2.4 kBd 7E1)
Paßwort: "info"	454 000 130 32 (DATEX-P)

Tabelle B.5: Mailboxadressen

B.8 "Der Netz-Knigge"

Der kleine "Netz-Knigge" versteht sich als Empfehlung und Mahnung zugleich. Als Empfehlung insofern, als daß einige einfache Tips helfen können, eine Menge Ärger zu vermeiden. Man sollte immer daran denken, daß man auf den Netzen — zum Glück! — nicht allein ist, und daß gegen Benutzer, die den Betrieb ungünstig beeinflussen, durchaus vorgegangen wird. Soviel als Mahnung.

- Fast alle Server-Programme verstehen den Befehl `help`. Es ist sehr empfehlenswert, beim Zugriff auf einen neuen Server diesen Befehl abzusetzen, um erste Informationen zur Bedienung zu erhalten. Meistens enthält die Antwort eine knappe Bedienungsanleitung und Hinweise darauf, wie weitere, spezifischere Informationen bestellt werden können.

- Nicht die Geduld verlieren, wenn die Antwort ein paar Tage auf sich warten läßt und nicht gleich dieselbe Bestellung noch einmal aufgeben. Die Nachricht oder die Antwort könnte noch in irgendeiner Warteschlange auf einem Zwischenknoten auf den Weitertransport warten. Außerdem sind die Server oft sehr stark belastet oder bearbeiten Aufträge nur zu bestimmten Zeiten. Doppelte Anforderungen und damit doppelte Antworten belasten das Netz unnötig.
- Aus demselben Grund sollte man vor der Bestellung von Dateien speziell über FTP- und `trickle`-Server prüfen, ob auf ASCII oder `binary`-Mode umgeschaltet bzw. uuencodiert werden muß. Es ist ärgerlich, wenn nach mehreren Tagen Wartezeit eine 1 MB-Datei zwar endlich eintrifft, aber vom mail-Programm völlig zerstört worden ist.
- Besonders bei der Bestellung größerer Dateien sollte man prüfen, welcher Knoten diese bereithält und davon den nächsten benutzen. Von dem persönlichen Vorteil abgesehen, daß die Antwort schneller da ist, wird dadurch die Belastung vieler Verbindungen und Rechner vermieden. Diese Strategie wird auch von den Netzwerkbetreibern mit den "`trickle`"-Servern (siehe dort) verfolgt.
- Last but not least: "Die Freiheit eines Einzelnen endet dort, wo die des anderen eingeschränkt wird". Das gilt auch für die Benutzung öffentlicher Netze. Die Kontrollen sind gering, lediglich kommerzielle Nutzung wie z.B. das Versenden von Werbung ist verboten. Dazu sollte man die lokalen Regelungen und Vorschriften beachten. Dennoch: Wer ständig hochauflösende 1 MB-Grafik-Dateien unzureichend bekleideter Frauen/Männer aus Neuseeland bestellt, wird sich auf Dauer bei den Betreibern der Zwischenknoten unbeliebt machen. Dies ist *kein* fiktiver Fall, sondern in den USA mittlerweile ein echtes Problem!

Anhang C

Informationen zu MS-DOS

C.1 Kommandos, Interrupts, Funktionen

[1]Unter OS/2 muß `chcp` für Real- und Protected-Mode getrennt gegeben werden.
[2]Unter OS/2 im Protected-Mode entfällt die Angabe der Hauptspeicherbelegung.
[3]Im Protected-Mode können mehrere Dateispezifikationen angegeben werden.

Befehl	Typ	Mode	Funktion
`ansi`	E	P	Zulässigkeit ANSI-Sequenzen anzeigen/ändern
`append`	E	R	Pfad für Daten-Dateien definieren
`assign`	E	R	Laufwerk logisch umbenennen
`attrib`	E	RP	Dateiattribute anzeigen/ändern
`backup`	E	RP	Dateien sichern
`break`	I	R	Überwachung Ctrl Break anzeigen/ändern
`chcp`	I	RP[1]	Codepage anzeigen/ändern
`ch(dir)`	I	RP	Aktuelles Verzeichnis anzeigen/wechseln
`chkdsk`	E	RP[2]	Daten über Laufwerk ausgeben
`cls`	I	RP	Bildschirm löschen
`cmd`	E	RO	OS/2-Kommandointerpreter starten
`command`	E	R	MS-DOS Kommandointerpreter starten
`comp`	E	RP	Dateien miteinander vergleichen
`copy`	I	RP	Dateien kopieren (und umbenennen)
`date`	I	RP	Datum anzeigen/ändern
`del/erase`	I	RP	Dateien löschen
`det(ach)`	I	P	Hintergrundprozeß erzeugen
`dir`	I	RP[3]	Dateiverzeichnis ausgeben
`diskcomp`	E	RP	Disketten miteinander vergleichen
`diskcopy`	E	RP	Disketten kopieren
`dpath`	I	P	`append`-Ersatz in OS/2
`exit`	I	RP	Kommandointerpreter verlassen
`fdisk`	E	RP	Partitions anzeigen/ändern
`find`	E	RP	Nach String in Datei suchen
`format`	E	RP	Diskette formatieren
`graftabl`	E	R	Zeichen für Grafikbildschirm laden
`helpmsg`	E	RPO	Erklärung zu letztem Fehler ausgeben
`join`	E	R	Laufwerk als Unterverzeichnis anhängen
`keyb`	E	RP	Tastaturbelegung ändern
`label`	E	RP	Diskettennamen anzeigen/ändern

Tabelle C.1: Kommandos unter MS-DOS/OS/2, Teil 1

Befehl	Typ	Mode	Funktion
m(k)d(ir)	I	RP	Unterverzeichnis anlegen
mode	E	RP	Ports konfigurieren und umleiten
more	E	RP	Eingabe wiederholen, evtl. warten
patch	E	RPO	Datei an best. Stellen ändern
path	I	RP	Suchpfade für Programme festlegen
print	E	RP	Datei in Druckspooler einreihen
prompt	I	RP[4]	Prompt-String festlegen
recover	E	RP	Dateien mit zerstörten Blöcken lesen
ren(ame)	I	RP	Dateien umbenennen
replace	E	RP	Erweiterte copy-Version
restore	E	RP	Mit backup gesicherte Dateien wieder einlesen
r(m)d(ir)	I	RP	Verzeichnis löschen
set	I	RP	Environment-Variablen anzeigen/ändern
sort	E	RP	Eingabe sortiert ausgeben
spool	E	P	Spoolprogramm für best. Ports installieren
start	I	P	Prozeß starten
subst	E	R	Verzeichnis als Laufwerk ansprechen
sys	E	RP	Betriebssystemdateien kopieren
time	I	RP	Zeit anzeigen/ändern
tree	E	RP	Verzeichnisstruktur anzeigen
type	I	RP	Datei ausgeben
ver	I	RP	Betriebssystemversion ausgeben
verify	I	RP	Verify-Modus anzeigen/ändern
vol	I	RP	Diskettenname anzeigen
xcopy	E	RP	Verbesserte copy-Version

Tabelle C.2: Kommandos unter MS-DOS/OS/2, Teil 2

In den Tabellen C.1 und C.2 sind alle Kommandos aufgeführt, die unter MS-DOS und OS/2 zur Verfügung stehen. Die beiden Betriebssysteme wurden in einer Übersicht zusammengefaßt, weil die Befehlssätze fast identisch sind, wenn man von den Befehlen absieht, mit denen man Programme quasiparallel abarbeiten lassen kann.

- Die erste Spalte enthält den Namen des Kommandos, wie er nach dem Prompt einzugeben ist.
- Die zweite Spalte gibt Auskunft darüber, ob das Kommando vom Typ '**I**ntern' oder '**E**xtern' ist.
- In der dritten Spalte ist angegeben, ob der Befehl im **R**eal-oder im **P**rotected-Mode ausgeführt werden kann. Ein '**O**' markiert Kommandos, die nur unter OS/2 verfügbar sind. Die Angabe '**PO**' wäre überflüssig, denn ein Kommando, das nur im Protected-Mode benutzt werden kann, ist damit automatisch nur unter OS/2 und nicht unter MS-DOS verfügbar.

[4] Real- und Protected-Mode werden unterschieden.

- In der vierten Spalte schließlich wird die Funktion des Kommandos kurz erläutert.

Nr.	Adresse	Funktion
Prozessor-Interrupts		
00_{16}	00000_{16}	divide-by-zero
01_{16}	00004_{16}	single step (trace)
02_{16}	00008_{16}	NMI (**n**on **m**askable **i**nterrupt)
03_{16}	$0000C_{16}$	breakpoint (1-byte-interrupt)
04_{16}	00010_{16}	overflow on `INTO`
05_{16}	00014_{16}	ROM BIOS print screen (286+: bound-check failed)
06_{16}	00018_{16}	invalid opcode
07_{16}	$0001C_{16}$	processor extension not available
Hardware-Interrupts		
08_{16}	00020_{16}	timer tick (286+: double fault)
09_{16}	00024_{16}	keyboard (286+: segment overrun)
$0A_{16}$	00028_{16}	reserved (286+: IRQ2 cascade, invalid task-state segment)
$0B_{16}$	$0002C_{16}$	COM2 (286+: segment not present)
$0C_{16}$	00030_{16}	COM1 (286+: stack segment overflow)
$0D_{16}$	00034_{16}	LPT2 (286+: general protection fault)
$0E_{16}$	00038_{16}	disk controller (286+: page fault)
$0F_{16}$	$0003C_{16}$	LPT1
BIOS-Interrupts		
10_{16}	00040_{16}	video driver
11_{16}	00044_{16}	equipment check
12_{16}	00048_{16}	conventional memory size
13_{16}	$0004C_{16}$	disk driver
14_{16}	00050_{16}	serial communications port driver
15_{16}	00054_{16}	cassette (286+: I/O subsystem extensions)
16_{16}	00058_{16}	keyboard driver
17_{16}	$0005C_{16}$	parallel port printer driver
18_{16}	00060_{16}	ROM BASIC
19_{16}	00064_{16}	bootstrap
$1A_{16}$	00068_{16}	real-time (CMOS) clock driver
$1B_{16}$	$0006C_{16}$	ctrl-break
$1C_{16}$	00070_{16}	timer-tick
$1D_{16}$	00074_{16}	(pointer on) video parameter table
$1E_{16}$	00078_{16}	(pointer on) floppy-disk parameters
$1F_{16}$	$0007C_{16}$	(pointer on) font (graphics characters)

Tabelle C.3: Interrupts, Teil 1

Die zweite Spalte gibt die Adresse des Eintrags in der Tabelle der Interruptvektoren an, der auf die zugehörige ISR verweist. Die Tabelle steht ab Adresse 0000:0000 im Speicher.

Nr.	Adresse	Funktion
MS-DOS-Interrupts		
20_{16}	00080_{16}	terminate process
21_{16}	00084_{16}	DOS-functions
22_{16}	00088_{16}	terminate handler address
23_{16}	$0008C_{16}$	CTRL-C handler address
24_{16}	00090_{16}	critical-error handler address
25_{16}	00094_{16}	absolute disk read
26_{16}	00098_{16}	absolute disk write
27_{16}	$0009C_{16}$	terminate and stay resident
28_{16}	$000A0_{16}$	internal (keyboard busy loop; undoc.)
29_{16}	$000A4_{16}$	internal (fast putchar; undoc.)
$2A_{16}$	$000A8_{16}$	internal (network redirector; undoc.)
$2B_{16}$	$000AC_{16}$	internal (`IRET`)
$2C_{16}$	$000B0_{16}$	internal (`IRET`)
$2D_{16}$	$000B4_{16}$	internal (`IRET`)
$2E_{16}$	$000B8_{16}$	internal (execute command)
$2F_{16}$	$000BC_{16}$	multiplex interrupt (`print`, `assign`, `share`, `append`...)
33_{16}	$000CC_{16}$	Microsoft mouse driver
⋮	⋮	⋮
40_{16}	00100_{16}	floppy-disk driver (relocated interrupt 13_{16})
41_{16}	00104_{16}	(pointer on) fixed disk parameters
42_{16}	00108_{16}	video driver (relocated interrupt 10_{16})
43_{16}	$0010C_{16}$	(pointer on) EGA, MCGA, VGA character table
44_{16}	00110_{16}	(pointer on) EGA font
45_{16}	00114_{16}	reserved
46_{16}	00118_{16}	(pointer on) secondary fixed disk parameters
⋮	⋮	⋮
$4A_{16}$	00128_{16}	real time clock alarm (`IRET`)
⋮	⋮	⋮
50_{16}	00140_{16}	real time clock (`IRET`)
⋮	⋮	⋮
67_{16}	$0019C_{16}$	LIM EMS driver
⋮	⋮	⋮
70_{16}	$001C0_{16}$	
⋮	⋮	redirected hardware interrupts 9-15
77_{16}	$001DC_{16}$	

Tabelle C.4: Interrupts, Teil 2

Nr.	Beschreibung
00_{16}	reset disk system
01_{16}	get disk system status
02_{16}	read sector
03_{16}	write sector
04_{16}	verify sector
05_{16}	format track
06_{16}	format bad track
07_{16}	format drive
08_{16}	get drive parameters
09_{16}	initialize fixed disk characteristics
$0A_{16}$	read sector long
$0B_{16}$	write sector long
$0C_{16}$	seek
$0D_{16}$	reset fixed disk system
$0E_{16}$	read sector buffer
$0F_{16}$	write sector buffer
10_{16}	get drive status
11_{16}	recalibrate drive
12_{16}	controller RAM diagnostic
13_{16}	controller drive diagnostic
14_{16}	controller internal diagnostic
15_{16}	get disk type
16_{16}	get disk change status
17_{16}	set disk type
18_{16}	set media type for format
19_{16}	park heads
$1A_{16}$	format ESDI drive

Tabelle C.5: Funktionen BIOS-Disk-Interrupt

Nr.	ab	Beschreibung
00_{16}	1.0	terminate process
01_{16}	1.0	character input with echo (from `stdin`, to `stdout`)
02_{16}	1.0	character output (on `stdout`)
03_{16}	1.0	auxiliary input (from `stdaux`)
04_{16}	1.0	auxiliary output (on `stdaux`)
05_{16}	1.0	printer output (on `stdprn`)
06_{16}	1.0	direct console I/O (from `stdin`, to `stdout`)
07_{16}	1.0	unfiltered character input without echo (from `stdin`)
08_{16}	1.0	character input without echo (from `stdin`; check CTRL-break)
09_{16}	1.0	display string (to `stdout`)
$0A_{16}$	1.0	buffered keyboard input (from `stdin`)
$0B_{16}$	1.0	check input status (of `stdin`)
$0C_{16}$	1.0	flush input and then input (`stdin`)
$0D_{16}$	1.0	disk reset
$0E_{16}$	1.0	select disk
$0F_{16}$	1.0	open file (FCB)
10_{16}	1.0	close file (FCB)
11_{16}	1.0	find first file (FCB)
12_{16}	1.0	find next file (FCB)
13_{16}	1.0	delete file (FCB)
14_{16}	1.0	sequential read (open FCB)
15_{16}	1.0	sequential write (open FCB)
16_{16}	1.0	create file (FCB)
17_{16}	1.0	rename file (special FCB)
18_{16}		reserved
19_{16}	1.0	get current disk
$1A_{16}$	1.0	set DTA address
$1B_{16}$	1.0	get default drive data
$1C_{16}$	2.0	get drive data
$1D_{16}$		reserved
$1E_{16}$		reserved
$1F_{16}$		get default DPB (undoc.)

Tabelle C.6: Funktionen DOS-Funktions-Interrupt, Teil 1

Die zweite Spalte gibt die DOS-Version an, ab der die Funktion zur Verfügung steht. Die Hinweise FCB und open FCB bzw. ASCIIZ[5] und `handle` geben an, auf welche Weise ein Dateiname übergeben und eine geöffnete Datei referenziert werden.

[5] ASCII-String, entsprechend "C"-Konvention mit Null-(Zero-)Byte abgeschlossen.

Nr.	ab	Beschreibung
20_{16}		reserved
21_{16}	1.0	random read (open FCB)
22_{16}	1.0	random write (open FCB)
23_{16}	1.0	get file size (FCB)
24_{16}	1.0	set relative record number (open FCB)
25_{16}	1.0	set interrupt vector
26_{16}	1.0	create new PSP
27_{16}	1.0	random block read (open FCB)
28_{16}	1.0	random block write (open FCB)
29_{16}	1.0	parse filename (FCB)
$2A_{16}$	1.0	get date
$2B_{16}$	1.0	set date
$2C_{16}$	1.0	get time
$2D_{16}$	1.0	set time
$2E_{16}$	1.0	set verify flag
$2F_{16}$	2.0	get DTA address
30_{16}	2.0	get MS-DOS version number
31_{16}	2.0	terminate and stay resident
32_{16}		get DPB (undoc.)
33_{16}	2.0	get or set break flag (4.0: and get boot drive)
34_{16}		get pointer to `in_dos`-flag (undoc.)
35_{16}	2.0	get interrupt vector
36_{16}	2.0	get drive allocation information
37_{16}		get or set switch character (undoc.)
38_{16}	2.0	get or set country information
39_{16}	2.0	create directory
$3A_{16}$	2.0	delete directory (ASCIIZ)
$3B_{16}$	2.0	set current directory (ASCIIZ)
$3C_{16}$	2.0	create file (ASCIIZ, handle)
$3D_{16}$	2.0	open file (ASCIIZ, handle)
$3E_{16}$	2.0	close file (handle)
$3F_{16}$	2.0	read file or device (handle)

Tabelle C.7: Funktionen DOS-Funktions-Interrupt, Teil 2

Nr.	ab	Beschreibung
40_{16}	2.0	write file or device (handle)
41_{16}	2.0	delete file (ASCIIZ)
42_{16}	2.0	set file pointer (handle)
43_{16}	2.0	get or set file attributes (ASCIIZ)
44_{16}	2.0	IOCTL (I/O control)
45_{16}	2.0	duplicate handle
46_{16}	2.0	redirect handle
47_{16}	2.0	get current directory
48_{16}	2.0	allocate memory block
49_{16}	2.0	release memory block
$4A_{16}$	2.0	resize memory block
$4B_{16}$	2.0	execute program (alias EXEC; ASCIIZ)
$4C_{16}$	2.0	terminate process with return code
$4D_{16}$	2.0	get return code
$4E_{16}$	2.0	find first file (ASCIIZ)
$4F_{16}$	2.0	find next file (ASCIIZ)
50_{16}		set PSP (undoc.)
51_{16}		get PSP (undoc.)
52_{16}		get list of lists (undoc.)
53_{16}		translate BPB to DPB (undoc.)
54_{16}	2.0	get verify flag
55_{16}		create PSP (undoc.)
56_{16}	2.0	rename file (ASCIIZ)
57_{16}	2.0	get or set file date and time (handle)
58_{16}	3.0	get or set allocation strategy
59_{16}	3.0	get extended error information
$5A_{16}$	3.0	create temporary file (ASCIIZ, handle)
$5B_{16}$	3.0	create new file (ASCIIZ, handle)
$5C_{16}$	3.0	lock or unlock file region (`share`; handle)
$5D_{16}$		rename file with wildcard (and others, undoc.)
$5E_{16}$	3.1	get machine name, get or set printer setup
$5F_{16}$	3.1	device redirection (ASCIIZ)

Tabelle C.8: Funktionen DOS-Funktions-Interrupt, Teil 3

Nr.	ab	Beschreibung
60_{16}		canonicalize path string (undoc.)
61_{16}		reserved
62_{16}	3.0	get PSP address
63_{16}	2.25	get DBCS lead byte table
64_{16}		reserved
65_{16}	3.3	get extended country information
66_{16}	3.3	get or set code page
67_{16}	3.3	set handle count
68_{16}	3.3	commit file (handle)
69_{16}		reserved
$6A_{16}$		reserved
$6B_{16}$		reserved
$6C_{16}$	4.0	extended open file (ASCIIZ, handle)
$6D_{16}$		
$6E_{16}$		
$6F_{16}$		
70_{16}		
71_{16}		
72_{16}		
73_{16}		
74_{16}		
75_{16}		
76_{16}		
77_{16}		
78_{16}		
79_{16}		
$7A_{16}$		
$7B_{16}$		
$7C_{16}$		
$7D_{16}$		
$7E_{16}$		
$7F_{16}$		

Tabelle C.9: Funktionen DOS-Funktions-Interrupt, Teil 4

C.2 ASSIGN, JOIN, SUBST: Interne Datenstrukturen

`assign`, `join` und `subst` sind externe MS-DOS-Kommandos, die den Zugriff auf Laufwerke und Unterverzeichnisse beeinflussen. Nach Aufruf eines der angeführten Kommandos stimmen spezifizierter Pfad und tatsächlicher Pfad nicht mehr notwendigerweise überein. Für einen Watcher wie `AVWatchF`, der die Zulässigkeit von Operationen anhand von Programm- und Dateinamen beurteilt, stellt dies ein echtes Problem dar. Maßgeblich sind die Fragen: Auf welcher Systemebene wird die Abbildung vorgenommen; d.h. auf welcher Ebene tauchen die tatsächlichen Laufwerke und Dateinamen wieder auf? Oder gibt es eine Möglichkeit, die Abbildung selbst nachzuvollziehen und umzukehren? Antworten darauf liefert eine Untersuchung der Arbeitsweise der einzelnen Kommandos. Hinweis: Alle Angaben beziehen sich auf MS-DOS 4.0.

assign. Mit `assign <logdrive> <physdrive>` wird dem logischen Laufwerk `logdrive` das reale Laufwerk `physdrive` zugewiesen (engl. to assign). Danach wird ein Aufruf für `logdrive` auf das Laufwerk `physdrive` umgeleitet.

Funktion. Beim ersten Aufruf installiert sich `assign` als TSR-Programm resident im Speicher und übernimmt den DOS-Funktionsinterrupt 21_{16}. Durch die Übernahme des Interrupts empfängt `assign` Funktionsaufrufe vor dem Kernel und liefert diesem Dateinamen mit bereits veränderter Laufwerksangabe. In einer internen Tabelle von `assign` (Offset 0103_{16} zu PSP) wird die aktuelle Zuordnung festgehalten. Als Tabellenindex dient die Laufwerksangabe, wobei `A:` der 0 entspricht. Der Tabelleneintrag gibt das statt dessen zu verwendende Laufwerk an, wobei hier aber die Nummer 1 dem Laufwerk `A:` zugeordnet ist.

Ein Watcher, der *vor* `assign` installiert wurde und Kernel-Aufrufe kontrolliert, bekommt die tatsächlichen Dateinamen geliefert. Ein *nach* `assign` installiertes Prüfprogramm dagegen, das deswegen vor `assign` in der Interruptkette steht, empfängt die Dateinamen vor der Umsetzung, was zu falschen Schlüssen führen kann. Da ein Watcher aber so früh wie möglich, vor allen Dingen vor jeglichen anderen Programmen gestartet werden sollte, ergeben sich durch `assign` keine Probleme.

join und subst. Wird nach der Eingabe von `join <physdrive> <path>` eine Datei angesprochen, deren Pfad mit `path` beginnt, wird der Zugriff auf das Laufwerk `physdrive` umgeleitet. `physdrive` wird quasi als Verzeichnis `path` an das Dateisystem "angehängt" (engl. to join). Nach der Eingabe von `subst <physdrive> <path>` werden Zugriffe auf das Laufwerk `physdrive` auf das Verzeichnis `path` umgeleitet. Das Laufwerk `physdrive` wird durch das Verzeichnis `path` ersetzt (engl. to substitute).

Funktion. `join` und `subst` installieren sich weder speicherresident noch übernehmen sie Interrupts oder verwalten interne Tabellen. Statt dessen wird eine MS-DOS-interne Datenstruktur manipuliert, deren Lage im Speicher ebensowenig dokumentiert ist wie ihr Aufbau. Nach eigenen Nachforschungen ist der Eintrag für ein Laufwerk unter DOS 4.0 wie in Tab. C.10 beschrieben aufgebaut.

Offset	Bedeutung
00_{16}	Je nach Mode (Offset 44_{16}): 40_{16}: aktuelles Verzeichnis 50_{16}: statt Laufwerk anzusprechendes Verzeichnis 60_{16}: Verzeichnis, unter dem das Laufwerk erscheint
44_{16}	Mode 40_{16}: normal 50_{16}: `SUBST` 60_{16}: `JOIN`
57_{16}	Ende Eintrag

Tabelle C.10: Aufbau Eintrag in interner Laufwerkstabelle

Die Adresse des ersten Eintrags erhält man durch Aufruf der undokumentierten, aber oft benutzten[6] Funktion 52_{16} "Get List of Lists" des DOS-Funktionsinterrupts [15]. Die Vermutung, daß diese Liste einen Zeiger auf die oben beschriebene interne Tabelle enthält, bestätigte sich bei mehreren Tests. An Offset 16_{16} steht der gesuchte FAR-Zeiger.

Da die Manipulation dieser Tabelle jenseits der Interruptebene wirksam ist, kann ein Watcher, egal zu welchem Zeitpunkt dieser installiert wurde, das tatsächliche Laufwerk nicht bestimmen. Eine Alternative bestünde darin, die Tabelle auszuwerten und sich dabei auf undokumentierte Strukturen zu verlassen, die sich u.U. von einer Betriebssystemversion zur nächsten verändern. Deshalb sei empfohlen, die sowieso nur selten verwendeten Kommando `join` und `subst` einfach aus dem System zu entfernen.

[6] Auch und gerade von DOS-Kommandos.

Anhang D

Abkürzungsverzeichnis

Abkürzung	Bedeutung
Abb.	Abbildung
am.	amerikanisch
bes.	besonders
Bsp.	Beispiel
engl.	englisch
etc.	et cetera (und andere)
evtl.	eventuell (möglicherweise)
ff	und folgende
ggf.	gegebenenfalls
i.allg.	im allgemeinen
i.d.R.	in der Regel
insg.	insgesamt
max.	maximal
min.	minimal
mind.	mindestens
o.ä.	oder ähnlich
rel.	relativ
s.	siehe
s.a.	siehe auch
s.S.	siehe Seite
sl.	slang (Umgangssprache)
sog.	sogenannt
Tab.	Tabelle
u.a.	unter anderem
u.U.	unter Umständen
undok.	undokumentiert
versch.	verschiedene
vollst.	vollständig
z.B.	zum Beispiel
z.T.	zum Teil
z.Zt.	zur Zeit

Anhang E

Glossar

Die Übersetzung der englischen Begriffe steht jeweils in Hochkommata “ ”; Querverweise sind mit ’→’ markiert.

asymmetrische Verschlüsselung: Ver- und Entschlüsselung erfolgen mit zwei verschiedenen, nicht voneinander ableitbaren Schlüsseln, von denen einer öffentlich bekannt sein kann (→Public Key) und der andere geheim sein muß (→Private Key).

AT: Kurzform von →IBM-AT; steht für **A**dvanced **T**echnology (“fortgeschrittene Technik”).

Authentifizieren: Eine Person sicher identifizieren, die Urheberschaft einer Nachricht sicher (rechtsverbindlich für z.B. Verträge) feststellen.

AUTOEXEC.BAT: →Batch-Datei, die von `command.com` nach einem Neustart automatisch ausgeführt wird.

Backslash (engl.): Der “Rückwärts-Schrägstrich” ’\’.

Batch-Datei (engl.): Datei, die Befehle zur Batch- (“Stapel-”) Verarbeitung, d.h. DOS-Befehle, enthält.

BIOS: Das BIOS ist das “allgemeine Ein-/Ausgabe-System” (**B**asic **I**nput **O**utput **S**ystem) eines →PCs, über das die Kommunikation mit Disketten- →Controllern, seriellen (z.B. Maus) und parallelen (z.B. Drucker) Schnittstellen abgewickelt wird.

Bootprogramm: Kurzes Programm im →Bootblock, das nach einem →Reset vom →Bootstrap-Loader geladen und ausgeführt wird. Lädt das eigentliche Betriebssystem.

Bootstrap-Loader: Das englische Äquivalent zu "Sich an den eigenen Haaren aus dem Sumpf ziehen" (Münchhausen) heißt "Lifting yourself by your own bootstraps" = "Sich selbst an den eigenen Schnürsenkeln hochheben". Lädt das →Bootprogramm.

Bootstrap-ROM: s. ROM-BIOS

Bootblock: Erster logischer Block einer →Diskette/→Festplatte, der ein kurzes →Bootprogramm enthält, welches das eigentliche Betriebssystem lädt.

Boot-ROM: s. ROM-BIOS

BDSG: **D**eutsches **B**undesdaten**s**chutz**g**esetz.

CERT: Die Eingreiftruppe für Computer-Notfälle auf dem INTERNET (**C**omputer **E**mergency **R**esponse **T**eam).

Closed-Shop-Betrieb (engl.): Betrieb eines Rechenzentrums als "geschlossener Laden", d.h. für Anwender nicht zugänglich.

CMOS-RAM: Elektronische Bausteine in CMOS-Technologie verbrauchen besonders wenig Strom und werden daher bei →ATs dazu benutzt, akkugepufferte Konfigurationsdaten (→Setup) auch bei ausgeschaltetem Rechner über längere Zeit zu speichern.

COMMAND.COM: Kommandointerpreter von →MS-DOS, den der Anwender zur Kommunikation mit dem Rechner benutzt. Führt Befehle aus der Kommandozeile aus und interpretiert Stapeldateien.

CONFIG.SYS: Datei zur System-Konfiguration, in der die zu ladenden →Gerätetreiber und andere Parameter wie die Anzahl der max. gleichzeitig offenen Dateien, Dateipuffer etc. definiert sind.

Controller (engl.): Hardware zur Ansteuerung von →Disketten- und →Festplatten-Laufwerken.

Datei-Spezifikation: [Laufwerk] [Verzeichnis] Dateiname. Spezifiziert eine oder, bei der Verwendung von →Jokerzeichen, evtl. mehrere Dateien.

DES: Der **D**ata **E**ncryption **S**tandard ist ein vom →NIST entwickeltes schnelles und sicheres Verfahren zur →symmetrischen Verschlüsselung, das Bestandteil des →FIPS ist.

Device Driver (engl.): Ein "Gerätetreiber" ist ein →speicherresidentes Programm zur Gerätesteuerung, das während des →Bootvorgangs in →MS-DOS eingebunden wird (s.a. →`config.sys`).

Diskette: s. Floppy Disk

Environment : Block mit "Umgebungs"-Variablen, die mit dem DOS-Kommando `set` gesetzt, verändert und gelöscht werden können (z.B. `path`). Werden durch `command.com` und bestimmte andere Programme ausgewertet.

DoD: Das **D**epartment **o**f **D**efense (Verteidigungsministerium) der USA.

EEPROM: Das **E**lectrically **E**rasable PROM ist wie das →EPROM nichtflüchtig, programmier- und löschbar, kann aber ohne spezielle Hardware (besondere Schreibspannung, UV-Lampe) auf elektrischem Wege verändert werden.

EPROM: Ein **E**rasable →PROM kann im Gegensatz zu einem →PROM mehrmals durch das Aufbringen elektrischer Ladungen programmiert und durch UV-Licht gelöscht werden.

EXE-Header: "Vorspann" einer `exe`-Datei, der Adressinformationen enthält. MS-DOS benötigt diese Angaben beim Laden eines Programms zur Berechnung von im Code enthaltenen, verschiebbaren Adressen ([14], INTEL Relocatable Object Module Formats).

Festplatte: s. Harddisk

File Server (engl.): Der File Server ("Datei-Dienst") stellt den einzelnen Stationen, die an ein Netzwerk angeschlossenen sind, Programme und Daten zu Verfügung, die alle benötigen (z.B. das Betriebssystem).

FIPS: Der **F**ederal **I**nformation **P**rocessing **S**tandard ist die Vorschrift für Informationsverarbeitung durch staatliche Stellen in den USA.

Floppy (Disk) (engl.): Die Floppy Disk ("schlappe Scheibe") ist ein flexibler, austauschbarer magnetischer Datenträger, der zumeist in den Formaten $3\frac{1}{2}$", $5\frac{1}{4}$" und 8" Zoll Anwendung findet (ein Zoll = ein Inch = 2.54cm).

GAO: (United States) **G**eneral **A**ccounting **O**ffice.

Gerätetreiber: s. Device Driver

Harddisk: Magnetisches, unflexibles Speichermedium ("harte Scheibe"), das hohe Positioniergenauigkeit zuläßt und damit große Speicherkapazitäten ermöglicht. Bei →PCs meist nicht austauschbar (sonst Wechselplatte).

IBM-PC: **P**ersonal **C**omputer ("Arbeitsplatzrechner") des Herstellers **I**ndustrial **B**usiness **M**achines Corporation. Hier auch stellvertretend für alle dazu kompatiblen Rechner.

Interrupt (engl.): (Programm-) "Unterbrechung", die durch Hardware oder Software ausgelöst wird. Die Ausführung des aktiven Programms wird gestoppt und an die Interrupt Service Routine (→ISR) übergeben. Danach wird an der Stelle der Unterbrechung fortgefahren.

ISR: **I**nterrupt **S**ervice **R**outine, die die →Interrupt-Anforderung bedient.

Jokerzeichen: Die Jokerzeichen '`*`' und '`?`' stehen für eine Reihe von Zeichen bzw. einzelne Buchstaben in einem Dateinamen. "`dir *.exe`" listet z.B. alle Dateien auf, die einen beliebigen Namen und die Erweiterung "`exe`" haben.

MCB: Speicherkontrollblöcke (**M**emory **C**ontrol **B**locks), die →MS-DOS zur Verwaltung des Arbeitsspeichers verwendet.

MIT: **M**assachusetts **I**nstitute of **T**echnology (bedeutende Universität in den USA).

MS-DOS: Das **M**icro**s**oft - **D**isc **O**perating **S**ystem ist ein Betriebssystem für →IBM-PCs.

NIST: **N**ational **I**nstitute of **S**tandards and **T**echnology.

NCSC: **N**ational **C**omputer **S**ecurity **C**enter.

NSF: **N**ational **S**cience **F**oundation.

Orange Book: Bezeichnung für die vom →NCSC herausgegebenen "Trusted Computer System Evaluation Criteria" (Kriterien zur Bewertung von sicheren Computersystemen).

Partition (engl.): Logisches Laufwerk ("Abteilung") einer →Festplatte. Ein Festplattenlaufwerk kann zwei oder mehr Partitions haben, die jede für sich einen Teil der Gesamtkapazität reservieren.

Patching (engl.): Gemeint ist das "Flicken" (Abändern) bestimmter Stellen, typischerweise nur einiger Bytes, eines Programms.

PC: s. IBM-PC

PC-DOS: IBM-Betriebssystem für →PCs, weitgehend identisch mit MS-DOS.

Platte: s. Harddisk

Power On Reset (engl.): Der Power On Reset ("Spannung-Ein"-Reset) wird beim Einschalten der Versorgungsspannung ausgelöst, um den Rechner zu initialisieren.

POST: Der **P**ower **O**n **S**elf **T**est ("Selbsttest nach dem Einschalten") führt eine Diagnose der Hardware und der Systemkonfiguration durch, initialisiert den Rechner und lädt das Betriebssystem (s.a. →Bootstrap-Loader).

Public Key: Der "öffentliche Schlüssel" eines →asymmetrischen Verschlüsselungsverfahrens ist jedem bekannt und wird zur Verschlüsselung oder Entschlüsselung benutzt.

Private Key: Der "private Schlüssel" eines →asymmetrischen Verschlüsselungsverfahren ist nur einer Person oder einer Gruppe von Personen bekannt und wird zur Verschlüsselung (→Authentifikation) oder Entschlüsselung benutzt.

PROM: Ein **P**rogrammable ROM ist ein →ROM, das durch elektrisches Durchbrennen von Widerstandsbrücken einmalig programmierbar ist.

PSP: Jedem Programm geht ein von MS-DOS beim Einladen generiertes **P**rogram **S**egment **P**refix ("Programmsegmentvorspann") voraus, in dem Informationen über und für das aufgerufene Programm gespeichert sind (z.B. ein Zeiger auf die →Environment-Variablen).

RAM: Ein **R**andom **A**ccess **M**emory (flüchtiger "Speicher mit wahlfreiem Zugriff") verliert seinen Inhalt beim Abschalten der Spannungsversorgung.

RAM-Disk: Ein Teil des Arbeitsspeichers (→RAM) wird dazu verwendet, ein Laufwerk zu simulieren, das sehr schnell ist, weil es keine Anfahrzeiten, Latenzzeiten und Kopfbewegungen gibt. Allerdings geht der Inhalt nach Abschalten der Betriebsspannung verloren.

Red Book: Bezeichnung für die vom →NCSC herausgegebenen "Trusted Network Evaluation Criteria" (Kriterien zur Bewertung von sicheren Netzwerken).

Reset (engl.): Zurücksetzen des Systems auf einen definierten Zustand, um einen Neustart durchzuführen.

ROM: **R**ead **O**nly **M**emories (nichtflüchtige "Nur-Lese-Speicher") behalten ihren Inhalt auch ohne Spannungsversorgung.

resident (engl., aber eingedeutscht): Gebraucht im Sinne von "speicherresident"; bedeutet: im Speicher anwesend. →TSR-Programme werden resident installiert, indem das Betriebssystem die Ausführung beendet, den belegten Speicher aber nicht freigibt.

ROM-BIOS: Dieser 64kB große nichtflüchtige Teil des Betriebssystems stellt wichtige Grundfunktionen (sektorweises Lesen, Schreiben etc. auf →Disketten) und den →Bootstrap-Loader zu Verfügung.

RSA: Von R. Rivest, A. Shamir und L. Adleman am →MIT entwickelter rel. langsamer Algorithmus zur →asymmetrischen Verschlüsselung. Sicher durch die Schwierigkeit, sehr große Zahlen in ihre Primfaktoren zu zerlegen (Basis des Verfahrens).

Setup (engl.): Daten zur Systemkonfiguration beim →Reset, die bei →ATs in einem nichtflüchtigen, aber veränderbaren →CMOS-RAM gespeichert sind. Kann mit spez. Programmen oder durch Drücken einer bestimmten Tastenkombination beim →Reset verändert werden.

symmetrische Verschlüsselung: Ver- und Entschlüsselung erfolgen mit dem gleichen Schlüssel, der deswegen geheim bleiben muß.

TSR-Programm: Programm, das →resident im Speicher liegt.

Urlader-: s. Boot-

validiert: Von (engl.) "to validate" = "für zulässig erklären": für eine bestimmten Zweck zugelassen.

Wildcards (engl.): s. Jokerzeichen

Write Protect (Tab) (engl.): Der Write Protect Tab ("Schreibschutzaufkleber") verdeckt einen kleinen Ausschnitt am Rand der Diskettenhülle, um so dem →Controller mitzuteilen, daß das Beschreiben der Diskette verboten ist.

XT: Kurzform von →IBM-XT: steht für **E**xtended **T**echnology ("erweiterte Technik").

Besondere Zeichen

'&': Der Adressoperator der Sprache "C".
Beispiel: "`&Puffer`" bedeutet "Adresse von Puffer".

$\mathbf{x}_{16}$: Zahl in sedezimaler ("hexadezimaler") Darstellung.
Beispiel: "$\texttt{AA}_{16}$" bedeutet "170" in dezimaler Schreibweise.

Abbildungsverzeichnis

1.1 Aktivierung und Residenz von Viren 10
1.2 Infektions-Typologie 12
1.3 Von `ViruScan` erkannte Viren 19

2.1 Wie kommt das Virus ins System? 29
2.2 Schutzzonen und Kontrollpunkte 60

3.1 Mögliche und tatsächliche Segmentierung des Adreßraums 70
3.2 Verhalten des Stack bei `call near/far` und `int` 74
3.3 NEAR- und FAR-Adressierung 77
3.4 Fehlerhafte Adressierung bei Offsetüber- und -unterlauf 79
3.5 Bearbeitung eines Software-Interrupts 80
3.6 Einklinken in Interrupts 81
3.7 Parameterübergabe via Stack 85
3.8 Der Aufbau von MS-DOS 95
3.9 Speicherverwaltung unter MS-DOS 102
3.10 Verwaister Environment-Zeiger 108

4.1 Ein-/Ausgabe `ChkSys` 146
4.2 Ein-/Ausgabe `Seal` 153
4.3 Abhängigkeit Prüfzeit von gelesenen Sektoren pro Zugriff 155
4.4 Aufbau Textform Datei-/Verzeichniseintrag bei `ChkState` 161
4.5 Ein-/Ausgabe `ChkState` 164
4.6 Dateiindizes und deren Verwendung bei `ChkState` 166
4.7 Ein-/Ausgabe `AVCopy/AVRename` 171
4.8 Systemarchitektur `AVCopy/AVRename` 174
4.9 Zeiger auf Zeiger auf Datenobjekt 196
4.10 Aufbau Startcode für das Speichermodell `TINY` (Turbo-C) 201
4.11 Ein-/Ausgabe `AVWatch*` 207

A.1 Zugriff auf die Referenzliste von `ChkState` 296

Tabellenverzeichnis

1.1 Cohen's Experimente auf Großrechnern . . . 17
1.2 Sicherheitsstufen des Orange Book . . . 23

2.1 Dateiattribute unter MS-DOS . . . 38

3.1 Bedeutung der Register . . . 72
3.2 Adressierungsarten . . . 75
3.3 Probleme beim Adreßvergleich . . . 78
3.4 Speichermodelle . . . 82
3.5 Werterückgabe bei "C"-Funktionen . . . 87
3.6 Namen und Deklarationen von Segmenten . . . 90
3.7 Gerätenamen . . . 98
3.8 Aufbau MCB (Memory Control Block) . . . 101
3.9 Aufbau "List of Lists" . . . 103
3.10 Aufbau PSP (Program Segment Prefix) . . . 106
3.11 `get_mcb_name`: Ermittle Name des Besitzers eines MCBs . . . 109
3.12 `get_owner_mcb`: Ermittle für Adresse zugehörigen MCB . . . 110
3.13 Aufbau PBR (Bootsektor) . . . 112
3.14 Aufbau MBR (Master Boot Record) . . . 113
3.15 Aufbau Partitions-Eintrag . . . 113

4.1 Sicherheitsrelevante Kommandos unter MS-DOS . . . 133
4.2 Relevante Funktionen DOS-Funktionsinterrupt 21_{16} . . . 135
4.3 Schutzbereiche eines Watchers . . . 136
4.4 Standardaufbau Funktionsbeschreibung . . . 140
4.5 Aufbau Attribut-Byte . . . 147
4.6 `scan_dir`: Dateisystem rekursiv durchsuchen . . . 148
4.7 `check_seal`: Überprüfe Programmsiegel . . . 158
4.8 Beispiel "Dateibestand" . . . 160
4.9 Beispiel "Vergleich von Dateibeständen" . . . 161
4.10 `get_phenotype`: Bestimme Phänotyp (einer Datei) . . . 178
4.11 `get_genotype`: Bestimme Genotyp (einer Datei) . . . 179
4.12 `norm_path2`: Normalisiere Pfad 2 . . . 183
4.13 `norm_path`: Normalisiere Pfad . . . 185
4.14 `intercom`: Kommunikation mit residentem Programm . . . 197

4.15 Datenzugriff von TSR-Programmen 205
4.16 Bitmuster erforderliche Rechte unter `AVWatchP` 213
4.17 Schutzmaßnahmen unter `Flushot` 218
4.18 Schutzmaßnahmen unter `AVWatchF` 220
4.19 `cmp_fspec`: Vergleiche Dateispezifikationen 221
4.20 Aufbau des Modus-Wortes in `set` (`isr_21`) 223
4.21 `add2log`: Füge Eintrag an Logdatei an 229

5.1 Viren-Spezialitäten 249

A.1 Get Current Disk (INT 21_{16}, Funktion 19_{16}) 258
A.2 Get Drive Data (INT 21_{16}, Funktion $1C_{16}$) 259
A.3 Set Interrupt Vector (INT 21_{16}, Funktion 25_{16}) 260
A.4 Parse Filename (INT 21_{16}, Funktion 29_{16}) 261
A.5 Get Date (INT 21_{16}, Funktion $2A_{16}$) 262
A.6 Get Time (INT 21_{16}, Funktion $2C_{16}$) 263
A.7 Terminate and Stay Resident (INT 21_{16}, Funktion 31_{16}) 263
A.8 Get Interrupt Vector (INT 21_{16}, Funktion 35_{16}) 264
A.9 Set Current Directory (INT 21_{16}, Funktion $3B_{16}$) 264
A.10 Get Device Information (INT 21_{16}, Funktion 44_{16}, Subfunktion 00_{16}) . . 265
A.11 Get Current Directory (INT 21_{16}, Funktion 47_{16}) 266
A.12 Release Memory Block (INT 21_{16}, Funktion 49_{16}) 267
A.13 Load and Execute (INT 21_{16}, Funktion $4B_{16}$) 267
A.14 Find First/Next File (INT 21_{16}, Funktion $4E/4F_{16}$) 268
A.15 Get Address of List of Lists (INT 21_{16}, Funktion 52_{16}) 269
A.16 Rename File (INT 21_{16}, Funktion 56_{16}) 270
A.17 Get or Set File Date and Time (INT 21_{16}, Funktion 57_{16}) 271
A.18 Get PSP Address (INT 21_{16}, Funktion 62_{16}) 272
A.19 Absolute Disk Read (INT 25_{16}) 272
A.20 Set Cursor Position (INT 10_{16}, Funktion 02_{16}) 273
A.21 Get Cursor Position (INT 10_{16}, Funktion 03_{16}) 274
A.22 Read Character and Attribute at Cursor (INT 10_{16}, Funktion 08_{16}) . . . 275
A.23 Write Character and Attribute at Cursor (INT 10_{16}, Funktion 09_{16}) . . . 276
A.24 INT 13_{16}: “Disk-Interrupt” 277
A.25 INT 16_{16}: “Keyboard-Interrupt” 278
A.26 Funktionsübersicht `AVSys.LIB` 280
A.27 Übliche Generatorpolynome 281
A.28 `block_crc`: Berechne CRC-Prüfsumme über Puffer 281
A.29 `make_crc_table`: Berechne CRC-Tabelle 282
A.30 `sig_crc`: Berechne CRC-Prüfsumme über Datei 282
A.31 `is_device`: Prüfe, ob Dateiname zu einem Gerät gehört 283
A.32 `message`: Gebe Nachricht auf Bildschirm aus 284
A.33 `tokenise`: Wandle Rechtestring in Rechtewort um 285
A.34 `add_path`: Stelle Dateinamen Startpfad voran 286
A.35 `cmp_fname`: Vergleiche Dateinamen 287
A.36 `comp_fspec`: Füge Pfad und Dateinamen zusammen 287

A.37 `fill_in`: Bilde Dateinamen auf Maske ab 288
A.38 `split_fspec`: Zerlege Dateinamen 289
A.39 `clean`: Bereinige String 290
A.40 `stricmpu`: Vergleiche Strings bis <Zeichen> 290
A.41 `strcpyu`: Kopiere String bis <Zeichen> 290
A.42 `stristr`: Suche String in String 291
A.43 `add2list`: Sortiere Text in Liste ein 292
A.44 `delete_list`: Gebe durch Liste belegten Speicher frei (lösche Liste) . . 292
A.45 `load_list`: Lese Textdatei zeilenweise in sortierte Liste ein 293
A.46 `select_p`: Prüfe, ob Dateiname in Liste enthalten ist 293
A.47 `arg_p`: Suche Kommandozeilenparameter 294
A.48 `get_arg`: Bestimme Index des n-ten Kommandozeilenparameters 295
A.49 `read_t_rights`: Lese und kompiliere Transportrechte 296
A.50 `get_ref`: Suche und lese Dateieintrag in Referenzdatei 298
A.51 `put_ref`: Schreibe Dateieintrag in Referenzdatei 298
A.52 `get_rights`: Überprüfe Rechtmäßigkeit einer Operation 300

B.1 Dateiendungen und zugehörige Pack-/Entpackprogramme 305
B.2 Befehlsauswahl `listserv` 306
B.3 Befehlsauswahl `trickle` 307
B.4 Befehlsauswahl `netserv` 309
B.5 Mailboxadressen 309

C.1 Kommandos unter MS-DOS/OS/2, Teil 1 312
C.2 Kommandos unter MS-DOS/OS/2, Teil 2 313
C.3 Interrupts, Teil 1 315
C.4 Interrupts, Teil 2 316
C.5 Funktionen BIOS-Disk-Interrupt 317
C.6 Funktionen DOS-Funktions-Interrupt, Teil 1 318
C.7 Funktionen DOS-Funktions-Interrupt, Teil 2 319
C.8 Funktionen DOS-Funktions-Interrupt, Teil 3 320
C.9 Funktionen DOS-Funktions-Interrupt, Teil 4 321
C.10 Aufbau Eintrag in interner Laufwerkstabelle 323

Wichtiger Hinweis:
Folgende Rubriken wurden noch nicht in die Literatursammlung aufgenommen:

- Buchbesprechungen
- Rubrik “Random Bits & Bytes”
- alle Zeitungsartikel

Literaturverzeichnis

[1] Literatur zur Programmerstellung

Programmieren in Assembler

[2] BRADLEY, David J.: *Programmieren in Assembler*
Carl Hanser Verlag, München Wien 1986
Grundlagen Assembler (80*)

[3] THIES, Klaus-Dieter: *PC/AT/XT Assembler Buch*
te-wi Verlag GmbH, München 1988
Grundlagen Assembler (80*); Handbuch zu MASM

Programmieren in C

[4] Borland (Herst.): *Turbo-C Benutzerhandbuch* und *Turbo-C Referenzhandbuch*
Borland 1990
Interna und Arbeit mit Turbo-C; Schnittstelle zu Assembler

[5] KERNIGHAN, Brian W. & RITCHIE, Dennis M.: *Programmieren in C*
Carl Hanser Verlag, München Wien 1983
Grundlagen C

Programmieren auf PCs unter MS-DOS

[6] ALTHAUS, Martin: *Das PC Profibuch*,
Sybex-Verlag GmbH, Düsseldorf 1988
Aufbau Master-Bootblock; Partitions

[7] Ara International Co., LTD (Herst.): *Optima/*1024 *VGA-Sync* (engl.)
Ara International Co., LTD. 1989
Funktionen des BIOS-INT 0x10 (Video)

[8] DUNCAN, Ray: *Advanced MS-DOS Programming* (engl.)
Microsoft Press 1989
Aufbau Master-Bootblock, Partitions

[9] GERDES, Martin: *Losgelassen - Wie man residente Programme wieder los wird*
in: [c't] 8/90, S.270-283
Entfernen von TSR-Programmen

[10] GOEBEL, Amy J.: *TRYST.DOC* (engl.)
über TRICKLE-Server, Directory "MSDOS.SYSUTL" 1988
Detailinformation zum Urladeprozeß auf IBM PCs

[11] IBM (Herst.): *Technical Reference* (engl.)
Industrial Business Machines Corporation 1985
Grundlagen Hardware, bes. Harddisk

[12] KAMIN, Jonathan: *MS-DOS Profibuch mit Harddisk-Management*
SYBEX-Verlag GmbH, Düsseldorf 1989
Bootvorgang; Organisation Boot-Block

[13] LAI, Robert S.: *MS-DOS-Device-Treiber*
Addison-Wesley Publishing Company 1986
Grundlagen Device Driver

[14] Microsoft Corporation (Herst.): *Microsoft MS-DOS 3.1 Programmierhandbuch (Programmer's Reference Manual) in englischer Sprache* (engl.)
Markt & Technik Verlag 1986
DOS-Interrupts; Device Driver

[15] SMODE, Dieter: *Das große MS-DOS-Profi-Arbeitsbuch*
Franzis-Verlag GmbH, München 1988
DOS-Interrupts; Aufbau Bootblock; Bootvorgang

[16] TANDON (Hrsg.): *Benutzerhandbuch*
Tandon Corp. 1985
Durch Setup steuerbares Bootverhalten

[17] WEITZ, Carl Markus & STILLER, Andreas: *Selbstfindung - Residente PC-Programme erkennen ihre Kopien*
in: [c't] 9/90, S.224 ff
Grundlagen TSR-Programme

[18] (Quelle: Server TRICKLE@DS0RUS1I): *ANSI-SYS.ARC*
über TRICKLE-Server, Directory "MSDOS.SYSUTL" 1988
ANSI-Sequenzen

Literatur zu Computeranomalien

Viren

Theoretische Untersuchungen, Experimente

[19] *Ernste Gefährdung oder verkannte Gefahr?: Virusprogramme*
[CP] 24 '86 S.100-105
Als Computerviren noch als Insider-Geheimnisse galten: Artikel der "Bayrischen Hackerpost" (BHP); Cohen's Arbeiten; APPLE II-Virus **zum abtippen**

[20] *Virus-Blockade*
[HC] 7 '86 S.16
FU Berlin sichert Großrechner speziell gegen Computerviren

[21] COHEN, Fred: *Computer Viruses: Theory and Experiments* (engl.)
in: [29], S. 297-310; ursprünglich in [C&S], 6/87, S. 22-35
Definition "Computervirus"; Beweis der nicht-Detektierbarkeit von Computerviren; Experimente zur Ausbreitung von Viren unter versch. Betriebssystemen

[22] COHEN, Fred: *On the Implications of Computer Viruses and Methods of Defense* (engl.)
in: [29], S. 331-341; ursprünglich in [C&S], 7/88, S. 167-184
Grundlagen; Experimente unter versch. Betriebssystemen; Detektion, biologische Analogien, Recovery, Immunisierung, Fehlertoleranz; zukünftige Strategien: Schutz der Integrität

[23] EVANS, Paul; SHAIN, Mike; GLISS, Hans & SOLOMONIDES, Costas: *Panel Discussion: Are Computer Viruses Here to Stay* (engl.)
[C&S] Vol. 9, No. 4 (1990) S.305-307
Diskussion auf der "Compsec '89" in London über Computerviren

[24] ZAJAC Jr., Bernard P.: *People: The "Other" Side of Computer Security* (engl.)
[C&S] Vol. 9, No. 4 (1990) S.301-303
Sicherheitsrisiko Personal

[25] ZAJAC Jr., Bernard P.: *The 1990s–What Will They Hold?* (engl.)
[C&S] Vol. 6, No. 6 (1990) S.503-507
Der Computer-Kriminelle von Morgen; mögliche Datenbank-Katastophen; Freiheit der Rede auf Computern; Datensicherung per DFÜ; Drahtlose WANs

Abwehr in Theorie und Praxis

[26] BRUNNSTEIN, Klaus: *Computer-Viren-Report*
WRS Verlag, München 1989
Grundlagen Computerviren

[27] BURGER, Ralph: *Das große Computer-Viren Buch*
Data Becker Verlag 1988
Grundlagen Computerviren; **Viren zum aptippen**

[28] COHEN, Fred: *Models of Practical Defenses Against Computer Viruses* (engl.)
in: [29], S. 343-354; ursprünglich in [C&S], 8/89, S. 149-160
Grundlagen; philosophische Hintergründe; Modelle zur Virendetektion in sicheren und unsicheren Systemen (durch Zeitmarken, Verschlüsselung); Einschränkungen bei der Detektion

[29] HIGHLAND, Dr. Harold Joseph: *Computer Virus Handbook* (engl.)
Elsevier Science Publishers Ltd. Oxford, England 1990
Aufsätze von Fred Cohen; Theorie zu Computerviren; Management; Gefahrenbeschränkung

[30] HUTT, Arthur E.: *Information Security – Is There More Than Hackers and Viruses?* (engl.)
[C&A] Volume 89.1 S.25-26, 29
-not yet reviewed-

[31] JONES, Scott K. & WHITE Jr., Clinton E.: *The IPM Model of Computer Virus Management* (engl.)
[C&S] Vol. 6, No. 5 (1990) S.411-418
Viren-Theorie; Vergleich mit biologischen Seuchen in der Landwirtschaft (Flora); Analogie der Maßnahmen: IPM (Integrated Pest Management) am Beispiel verschiedener Viren; Rechenbeispiele

[32] MUßTOPF, Günter (Hrsg.): *Trojanische Pferde, Viren und Würmer: Eine ernstzunehmende Gefahr für PC-Anwender?* (Begleitmaterial zum Film des SWF)
perComp-Verlag, Hamburg 1989
Grundlagen Computeranomalien, Typen, Funktion, Gefahren

[33] POZZO, Maria M. & GRAY, Terence E.: *An Approach to Containing Computer Viruses* (engl.)
in: [29] S.319-329; ursprünglich in [C&S], 6/87, S. 321-331
Theoretische Grundlagen zur Eindämmung von Viren; Detektion über Verschlüsselung; Stärken und Schwächen; Risikoklassen

[34] RÖSTEL, Nobert (Projektleiter); BAPPERT, Dagmar; DIETER, Stefan; SCHUBERT, Michael; TELGHEIDER, Thomas; TREBER, Christian: *Computerviren*
Studienarbeit zur Vorlesung "Projektmanagement" WS'89/'90 an der FH Fulda. Nicht öffentlich zugänglich (evtl. über Herrn Gillner, FH Fulda)
Grundlagen Computerviren

[35] SCHÖNEBURG, Eberhard; HEINZMANN, Frank & NAMYSLIK, Frank: *Computerviren*
Markt & Technik Verlag AG, Haar 1990
Passiver und vorbeugender Schutz

[36] SIMON, Hans J.: *Virenjagd per Grips - Computerviren ohne 'Impfstoffe' und 'Killerprogramme' aufspüren und beseitigen*
[c'tS] 5 1990 S.218-230
Mit 'konventionellen Waffen' (PCTools, Norton Utitilities etc.) Viren entdecken und bekämpfen

[37] STOLLER, Jack: *Virus Protection: The Security Officer's Role* (engl.)
[C&A] Volume 88.4 S.18-19
-not yet reviewed-

[38] WYK, Ken van (Editor Virus Diskussions-Forum): *VIRUS-L Digest* (engl.)
Distribution über BITNET mit Electronic Mail durch "VIRUS-L@IBM1.CC.LEHIGH.EDU". Bezugsquelle hier: "LISTSERV@LEHIIBM1.bitnet".
Aktuelle Information über gesamte Virenproblematik

[39] ZAJAC Jr., Bernard P.: *Computer Viruses: Can they be Prevented?* (engl.)
[C&S] Vol. 9, No. 1 (1990) S.25-31
Historie der Computerviren; Funktion, Abwehr, Detektion, Recovery

[40] *Gefahr Public Domain: Viren und ihre Bekämpfung*
[KES] '91/1 S.30-33
Computeranomalien-Typen; Vorkehrungen gegen Viren; Frühwarnsysteme; Virenversicherung

[41] *Wie kann man sich vor Viren schützen?*
[PP] 9 1989 S.120-122
Datensicherung, Verschleierung, Prüfsummenberechnung, codierte Speicherung und andere Verfahren

[42] *Procedures To Reduce The Computer Virus Threat* (engl.)
in: [29] S.279-293
Zuverlässige Quellen; Verwendung von Servern; organisatorische Aspekte; Detektion ohne Hilfsmittel; Verhaltensmaßregeln zur Risikoverminderung; Recovery; Katastrophenplan

[43] ABEL, Horst & Schmölz, Werner (Siemens AG): *Sicherheit mit dem System MX 300/MX 500*
[KES] KES Sonderheft in Zusammenarbeit mit Siemens
Planung der Systemsicherheit von *Installationsanforderungen* über *Betrieb* bis *Katastrophenvorsorge*

[44] AL-DOSSARY, Ghannam M.: *Computer Viruses Prevention and Containment on Mainframes* (engl.)
[C&S] Vol. 9, No. 2 (1990) S.131-137
Funktion von Computerviren; Typen, Abwehr

[45] HAHN, Mark: *Using CA-ACF2 and CA-TOP SECRET to Protect Your System* (engl.)
[C&A] Volume 88.4 S.15, 32
Systemschutz unter dem Aspekt Computerviren (eigentlich Würmer)

[46] KING, Martin: *Protecting Networks from Computer Viruses* (engl.)
[C&A] Volume 89.1 S.27-29
-not yet reviewed-

[47] KING, Marty: *Solving The Virus Problem On Your MVS Systems* (engl.)
[C&A] Volume 89.2 S.11-13
Tips zum Schutz von MVS-Libraries mit "CA-ACF2" und "CA-TOP SECRET"

[48] KING, Martin & BUER, Bill: *Viruses and Related Mechanisms in MVS* (engl.)
[C&A] Volume 88.4 S.14, 23-29
Definition Virus; Vergleich und Einschätzung der Bedrohung auf PCs/Mainframes; Definition und Bewertung weiterer Computeranomalien

Fälle konkreter Verseuchungen

[49] DAVID, Jon: *The Novell Virus* (engl.)
[C&S] Vol. 6, No. 7 (1990) S.593-599
Beschreibung des Versuchs; Testumgebung; Testergebnisse auf LANs

[50] HERSCHBERG, I.S. & PAANS, R.: *Friday the 13th: Facts and Fancies* (engl.)
[C&S] Vol. 9, No. 2 (1990) S.125-129
Bedrohung durch Computerviren am Beispiel von "Friday the 13th"; Fähigkeiten von Viren; Gegenmaßnahmen; Problem der nicht-Detektierbarkeit

[51] HERWEG, Ralf: *Praxisbericht: Virenverbreitung über PC-Schulungszentrum*
[KES] '90/1 S.22-23
Virenverbreitung ("Vienna-Virus") bei PC-Einführungsseminar in Köln

[52] *Vorsicht Viren!*
[STM] 6 '89 S.96
"SIGNUM-Virus" auf "ST-Digital Diskette Nummer 2" (Computec Verlag)

[53] *Weltweit befürchten die Experten heute Angriff von Computerviren*
[WELT] 1989-10-12
Entwicklung der Computerviren; Panik um das "Datacrime"-Virus

[54] *Computer-Viren: RWTH Aachen verschickte verseuchte Disketten*
[KES] '90/2 S.100-101
Akademisches Auslandsamt der RWTH Aachen verschickte unwissentlich mit dem "1701"- ("Herbstlaub"-)Virus verseuchte Software

[55] *Virus-Warnung: WHALE und Variante von Jerusalem-B*
[KES] '90/6 S.416
Jon David isoliert "JERUSALEM-B"-Variante, die Schutzeinrichtungen der Novell Netware umgeht; ein sophisticated virus: "WHALE"

Produkte und Produktbesprechungen

[56] *Keine Angst vor Viren?*
[PP] 9 1989 S.35-41
Produktübersicht/Vergleich Check- und Recovery-Programme

[57] HÜRTEN, Robert: *Marktübersicht: Hard- und Software zur PC-Sicherheit*
[KES] '90/3 S.171-194
Umfangreiche und ausführliche Marktübersicht

[58] *Computer-Viren: Testbericht: Suchprogramme*
[KES] '88/6 S.347-349
Prüfsummenprogramme im Test: "Alteration Searcher" (Data Becker), "PC-Checkup" (Gliss & Herweg GmbH), "Vaccine" (Sophos Ltd.); Vergleichsprogramme für Apple ("VIRUSRX") und Großrechner ("COMPAREX" von Sterling Software)

[59] (Rubrik "aktuell"): *Alarm für AT's*
in: [c't] 9/90, S.18
Bootschutz durch Hardware

[60] WEITZENDORF, Thomas (Teme, USA): *Kurztest: PCBoot: Kompaktes Tool zur Datensicherheit*
[KES] '90/6 S.414-415
Test "PCBoot 2.31" (Concord-Eracom, Wildberg); Zugriffsschutz und Datenverschlüsselung

[61] WETTMANN, Hartmut (Bonn): *Für Sie ausprobiert: SAVEDIR: Guter Schutz*
[KES] '90/5 S.341-343
Test "SAVEDIR" (Andreas Müller Software, Berlin)

[62] *Meldungen: Neues Anti-Viren-Programm*
[KES] '90/3 S.207
Besprechung von "Virus Block 3.0" (Expert-Informatik, Berlin)

Trojaner

[63] *Aids-Aufklärung mit verhängnisvollen Extras*
[WELT] 1989-12-20
Der "AIDS"-Trojaner: Beschreibung der Funktion; Erkenntnisse des BKA; Erforschung durch internationale Gruppen

[64] DIERSTEIN, Rüdiger: *Manipulations-Programme: Das Panama- oder AIDS-Programm*
[KES] '90/1 S.4-14
Der "AIDS"-Trojaner: detaillierte Besprechung

[65] *AIDS/PANAMA-Fall: Absender ermittelt*
[KES] '90/2 S.104
US-Bürger verhaftet: Dr. Joseph W. Popp ist wahrscheinlicher Initiator des "AIDS"-Trojaners

Würmer

[66] *Auf die Knie*
[SP] 1989-11-07 S.294-296
Der "INTERNET-Worm": Historie der Ausbreitung; Erforschung; Abwehrmöglichkeiten

[67] *"Die großen Systeme reizten Robert"*
[SP] 12/88 S.252-265
Der "INTERNET-Worm": Entstehungsgeschichte; Ausbreitung; Erforschung durch Army,FBI, CIA und NSA; Funktion (Lücken in UNIX); Grundlagen Computeranomalien

[68] *Meldungen: Morris verurteilt*
[KES] '90/2 S.102
Robert T. Morris ("INTERNET-Worm") schuldig gesprochen (mögl. Strafe: $250000 Geldstrafe und 5 Jahre Haft)

Artikel zum Thema Computersicherheit
PC-Sicherheit

[69] GLISS, Hans (Pulheim): *Sicherheits Enquête 1990: Arbeitsplatzrechner: Nachholbedarf*
[KES] '90/3 S.166-169
Umfrage über betrieblich eingesetzte Kontrollmaßnahmen (welche Maßnahmen, welche Bereiche, Kontrolle, Sicherheitsausbildung etc.

[70] *Grundlagen: Einstieg in die APC-Sicherheit*
[KES] '91/1 S.5-9
Spezielle Probleme der IDV (Individuellen Datenverarbeitung); Betriebssysteme der IDV: MS-DOS, OS/2, UNIX; Datensicherung- und Schutz

[71] *Sicherheit im Stand-Alone-Einsatz: Maßnahmen gegen Mißbrauch*
[KES] '91/1 S.34-35
Single- und Multi User-Betrieb; Sicherheitsmaßnahmen

[72] *Sichere APC-Netze: Vorteile für Server-Konzepte*
[KES] '91/1 S.36-42
Vorteile und Sicherheitsmaßnahmen in Netzen

[73] *Zehn Grundregeln der APC-Sicherheit: Kontrollbereiche des Datenschutzes*
[KES] '91/1 S.53-54
Erfüllung der Auflagen des BDSG (Bundesdatenschutzgesetzes); Organisation der Kontrollen; Maßnahmen zur deren Durchführung

[74] *Verteilte Datenverarbeitung: Datenschutzbeauftragte zum Thema PC-Sicherheit*
[KES] '91/1 S.372-373
Personalauswahl, Empfehlungen

[75] *Materielle Sicherheit: Installationsanforderungen beim APC-Einsatz*
[KES] '91/1 S.14-17
Übersicht Maßnahmen zur materiellen (physischen) Sicherheit

[76] *Sicherer DV-Betrieb: Regeln gegen Datenverlust*
[KES] '91/1 S.18-30
Zugangssperren für APC; System- und Benutzerverwaltung; Zugriffssicherung; Programmgfreigabe; Datensicherung, -Verwaltung; Entsorgung; Recovery

[77] GLISS, Hans & HERWEG, Ralf (Gliss und Herweg GmbH, Pulheim): *PC-Einsatz in Großunternehmen: Richtlinien für dezentralisierte Sicherheitsverantwortung*
[KES] '90/6 S.403-404
Wege zur PC-Sicherheit ohne zentrale Regelung der DV-Sicherheit

Sicherheit auf Minis und Großrechnern

[78] WETTMANN, Hartmut (Bonn): *Grundlagen: Wie sicher ist UNIX?*
[KES] '90/3 S.199-202
Überblick und Diskussion der Zugriffssicherungen unter UNIX

[79] SCHRAMM, Christof (Prüfungsstelle des Bayrischen Sparkassen- und Giroverbandes, München): *Revision: Nachtrag: Verbesserung der Sicherheit im MVS*
[KES] '90/3 S.203-205
Diskussion der Sicherheit und Anwendung von APFs (Autorized Program Facility)

[80] *Aktuell: VMS: Erneut Sicherheitslücke*
[KES] '90/6 S.407
Sicherheitsrisiko Kommando "ANALYSE/PROCESS-DUMP"

[81] *Eilmeldung: DEC-Windows ermöglicht unautorisierten Zugriff*
[KES] '91/1 S.13
Der Window-Manager, der von Windows mit System-Rechten aufgerufen wird, kann selbst gewählt **und erstellt** werden

Validierung von Software und Systemen

[82] *IT-Sicherheitszertifikate: ZSI: Evaluations-Handbuch liegt vor*
[KES] '90/5 S.348-349
Handbuch der ZSI (Zentralstelle für Sicherheit in der Informationstechnik) zur Bewertung der Sicherheit von Produkten der Informationstechnik (IT) liegt vor

[83] DIERSTEIN, Rüdiger (DLR Oberpfaffenhofen): *Kommentar: Amerika und die europäischen IT-Sicherheitskriterien*
[KES] '90/6 S.405-407
Vergleich "orange book" des DoD (Department of Defense) mit den europäischen IT-Sicherheitskriterien

[84] *Sicherheitskriterien: BSI bewertet DV-Produkte*
[KES] '91/1 S.50-52
Veröffentlichung von Kriterien des BSI (Bundesamtes für die Sicherheit von IT-Systemen); Verzeichnis von Sicherheitsprodukten

Ergänzende Literatur

[85] BRANDIS, Henning & OTTE, Hans Jürgen: *Lehrbuch der Medizinischen Mikrobiologie*
Gustav Fischer Verlag, Stuttgart New York 1984
Grundlagen Immunologie; allgemeine Virologie

[86] DIERSTEIN, Rüdiger (DLR Oberpfaffenhofen): *Kommentar: Viren als Aprilscherz*
[KES] '90/2 S.98-100
Wettbewerb um die Erstellung eines Computerviruses (Ikarus-GmbH); Ethik in der Informatik ("Eid des Turing"

[87] EICKER, Hermann Josef & MEIER, Rolf: *Computerkriminalität im Vormarsch: Hacker, Crasher, Datendiebe*
[WS] 10/89 Beilage "PC-Profi"
Einschätzung der Szene; der KGB-Hack; Raubkopieren; rechtliche Grundlagen

[88] SEIDEL, Ulrich (Gesellschaft für Mathematik und Datenverarbeitung (GMD), St. Augustin): *Übertragungssicherheit: Sicherer als Unterschriften: Elektronische Signatur*
[KES] '90/5 S.315-319
Vorstellung und Diskussion der Sicherheit von private- und public key- Verfahren; rechtliche Aspekte

[89] *Computer-Viren: Schäden in USA und Europa*
[KES] '88/6 S.344-346
Morris' "INTERNET-Worm" (Beschreibung der Funktion); Schäden durch Computerviren; Fälle von verseuchter Original-Software; Zerstörung von Hardware; Schutzmechanismen

[90] *Meldungen: Nach ABC-Waffen jetzt D-Waffen?*
[KES] '90/4 S.278
Pentagon-Projekt zur Entwicklung von Computerviren als Waffe; Diskussion

[91] *Stellenanzeige der Firma "G DATA": Softwarespezialisten für Computerviren (ATARI ST und IBM PC) gesucht*
[unbekannt]
Stellenanzeige "Softwarespezialisten für Computerviren"

[92] (Quelle: Server TRICKLE@DS0RUS1I): *SUIT.TXT* (engl.; nicht mehr verfügbar) über TRICKLE-Server, Directory "MSDOS.TROJAN-PRO"
Anklageschrift: Einschleusung eines Virus in ein BBS (Bulletin Board System)

[93] *Wenn der PC Schnupfen bekommt*
[unbekannt] 1989-10-11 S.10-11
Funktion von Computerviren; Besprechung zu Ralf Burger's Buch "Das große Computerviren-Buch"

[94] Der Bundesbeauftragte für Datenschutz (Hrsg.): *Bürgerfibel Datenschutz*
Roeder rund um den Druck, Niederkassel 1987
Rechtliche Grundlage für u.a. Log-Dateien

[95] *Praxis-Tip: Der Benutzerservice - eine Stelle der Koordination*
[KES] '91/1 S.10-13
Argumente für Hardware- und Softwarekataster; Verfahren für Software-Freigabe; rechtliche Vorschriften

[96] *Praxis-Tip: Wie schütze ich meine Daten?*
[CP] 24 '86 S.82-83
BDSG; Kontrollmaßnahmen

[97] *Wirtschaftsspionage: KGB weiter aktiv*
[KES] '90/2 S.103-104
Trotz STASI-Auflösung betreibt KGB Spionage über kompromittierende elektromagnetische Abstrahlung von Rechnern

[98] GERHARD, Harald: *Praxis-Tip: Geheimcode sinnvoll wählen*
[KES] '90/5 S.310-311
Paßwort-Auswahl; Methoden zur Erhöhung der Sicherheit

[99] *Kryptographie: DES und RSA Überarbeitungsbedürftig*
[KES] '90/5 S.312-14
Vorstellung und Diskussion der Sicherheit der DES- und RSA-Verschlüsselungs-Verfahren

[100] *Bundesrechnungshof: Sicherheitsmängel in staatlichen Rechenzentren*
[KES] '90/5 S.344-345
Verstöße gegen das Bundesdatenschutzgesetz; Sabotage bei TELEKOM

[101] *Begriffe und Gesetze: Gefahren durch Computerkriminalität*
[KES] '91/1 S.55-57
Definition: Betrug, Spionage, Sabotage, Mißbrauch; Gesetze; Literatur

Zeitschriften

[CP]
Computer Persönlich

[C&S]
Computers & Security
Elsevier Science Publishers Ltd.

[c't]
c't - magazin für computertechnik
Verlag Heinz Heise GmbH & Co KG, Hannover

[HC]
Happy Computer
Markt & Technik Verlag AG, Haar

[KES]
Kommunikations- und EDV-Sicherheit
aktuelle Information über Datenschutz- und Sicherheit

[PP]
PCPLUS

[C&A]
Security & Audit News
Computer Associates World Headquarters 711 Steward Avenue Garden City, NY 11530-4787
Information zu Computer Associates-Produkten

[SP]
Der Spiegel

[STM]
ST Magazin
Markt & Technik Verlag AG, Haar

[WELT]
Die Welt
Axel Springer Verlag

[WS]
WirtschaftsSpiegel
Deutscher Sparkassenverlag GmbH, Stuttgart

Anschriften

Markt und Technik, Hans-Pinsel-Straße 2, W-8013 Haar